Computer Systems: A Programmer’s Perspective 计算机系统

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

Linking

- **Linking：**
  - **Motivation**
  - **What it does**
  - **How it works**
  - **Dynamic linking**
- **Position-Independent Code (PIC)**
- **Case study: Library interpositioning**

Outline

- **链接技术核心基础**
  - 链接的动机与核心价值
  - 目标文件的分类
  - 链接器的核心功能
- **符号解析与重定位**
- **代码重用**
  - 静态链接库
  - 动态链接库
- **链接器的应用**
  - 程序的模块化构建
  - 符号控制与冲突消解
  - 代码复用
  - 调试与监控：库插桩技术

Example C Program

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

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

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

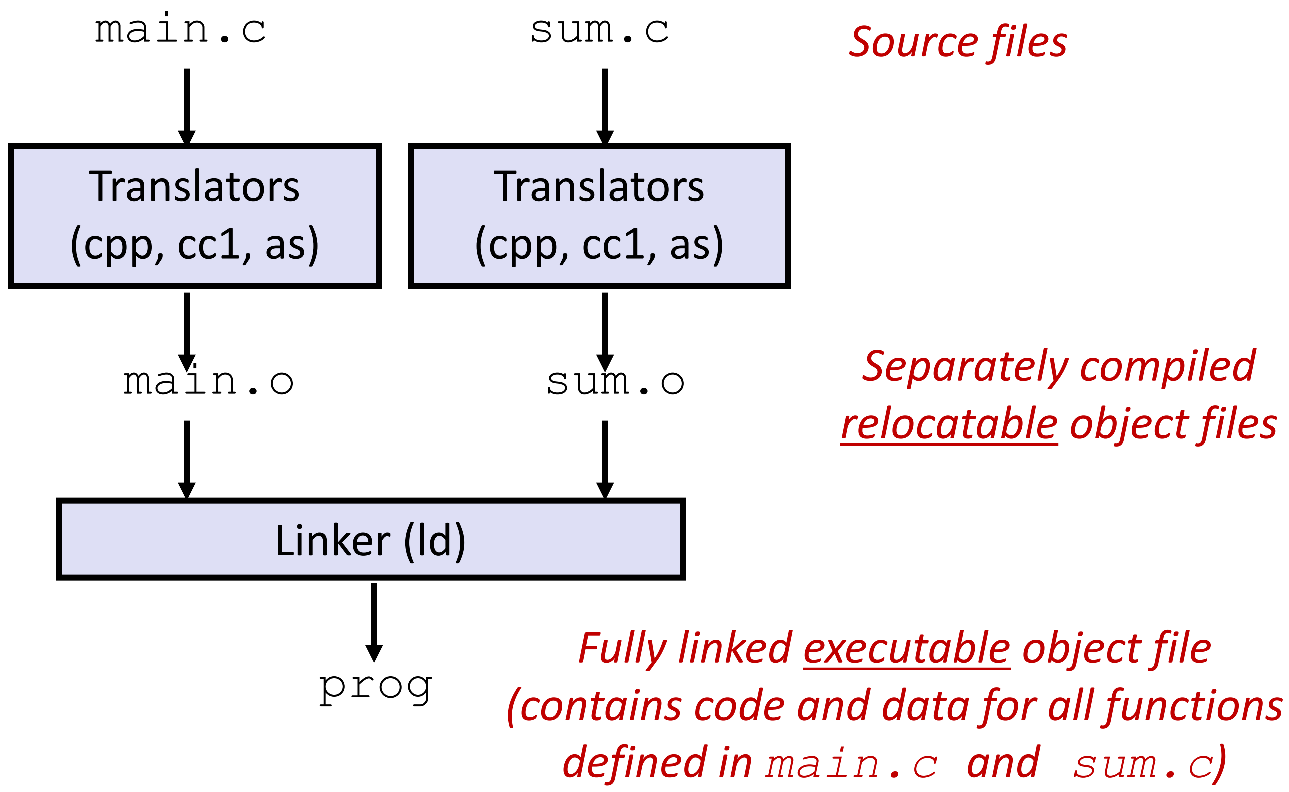
main.c

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

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

sum.c

Static Linking

- **程序使用编译器编译和链接:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



main.c
sum.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?

- **链接(linking)是将各种代码和数据片段收集并组合成为一个单一文件的过程。**
  - 链接过程由Linker来完成

- **Reason 1: 应用程序的模块化(Modularity)构建**
  - 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: 应用程序的高效化(Efficiency)构建**
  - 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
    - 公共的常用功能可以被集中到一个单独的文件中...
    - Option 1: Static Linking
      - 可执行文件和运行时内存映像只包含它们实际使用的库代码
    - Option 2: Dynamic linking
      - 可执行文件不包含库代码
      - 在执行过程中，库代码的单个副本可以被所有执行进程共享

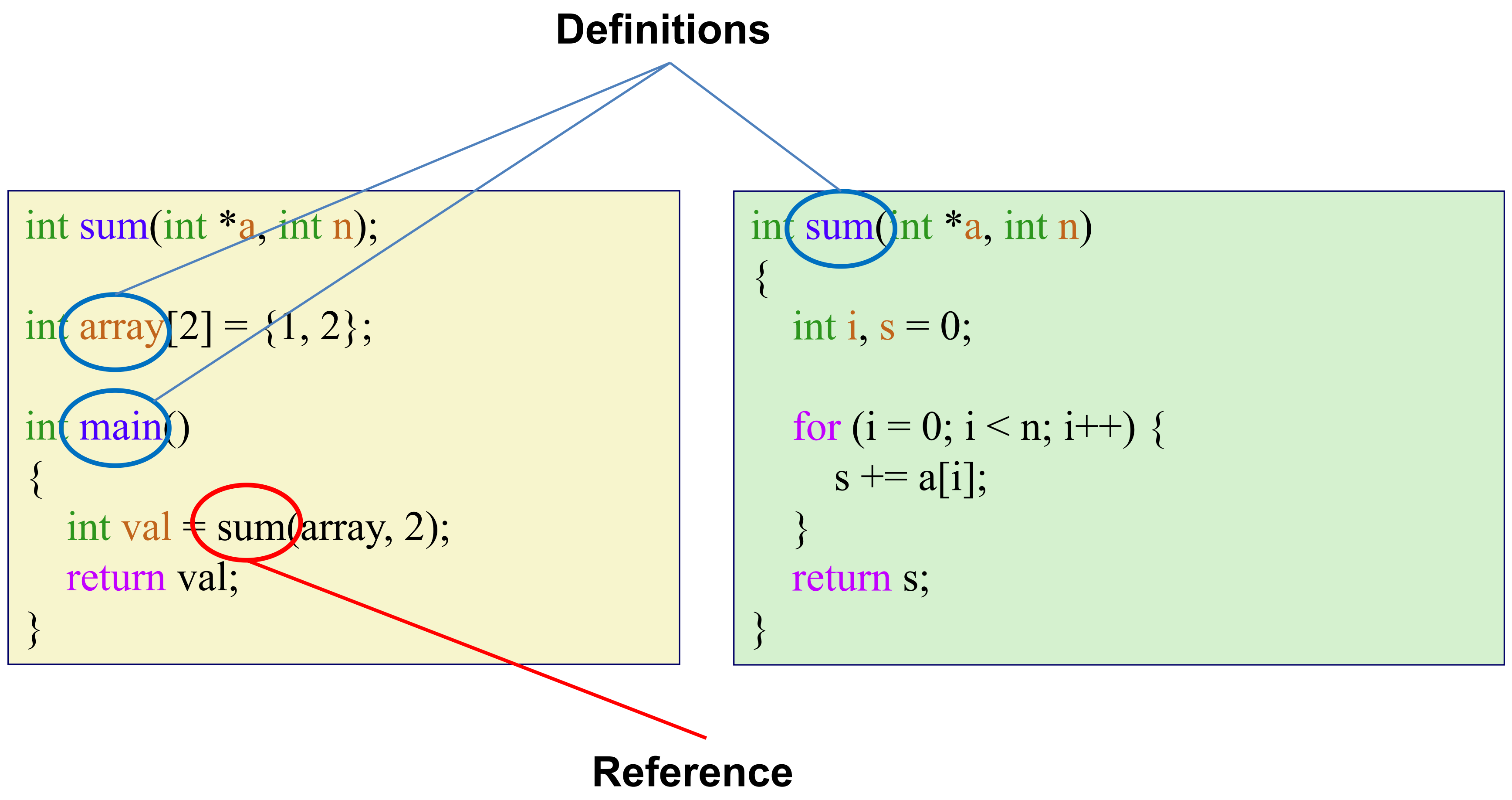
What Do Linkers Do?

- **Linker主要完成两步工作：符合解析和地址重定位**
- **Step 1: 符号解析(Symbol resolution)**
  - 程序定义和引用符号(*symbols*) (global variables and functions):

```
void swap() {…}    /* define symbol swap */
swap();            /* reference symbol swap */
int *xp = &x;      /* define symbol xp, reference x */
```

  - 符号定义存储在由汇编器生成的对象文件中的符号表(*symbol table*)中.
    - Symbol table is an array of `structs`
    - Each entry includes name, size, and location of symbol.
  - 符号解析步骤中，链接器(linker)将每个符号引用仅与一个符号定义关联起来 **(associates each symbol reference with exactly one symbol definition.)**

Symbols in Example C Program



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

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

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

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

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

What Do Linkers Do? (cont)

- **Step 2: 重定位(Relocation)**
  - 将不同.o 文件中的代码段和数据段合并到一个汇聚文件的代码和数据段中
  - 将符号从.o文件中的相对位置移动到可执行文件中的最终绝对内存位置
  - 更新所有对这些符号的引用以反映它们的新位置

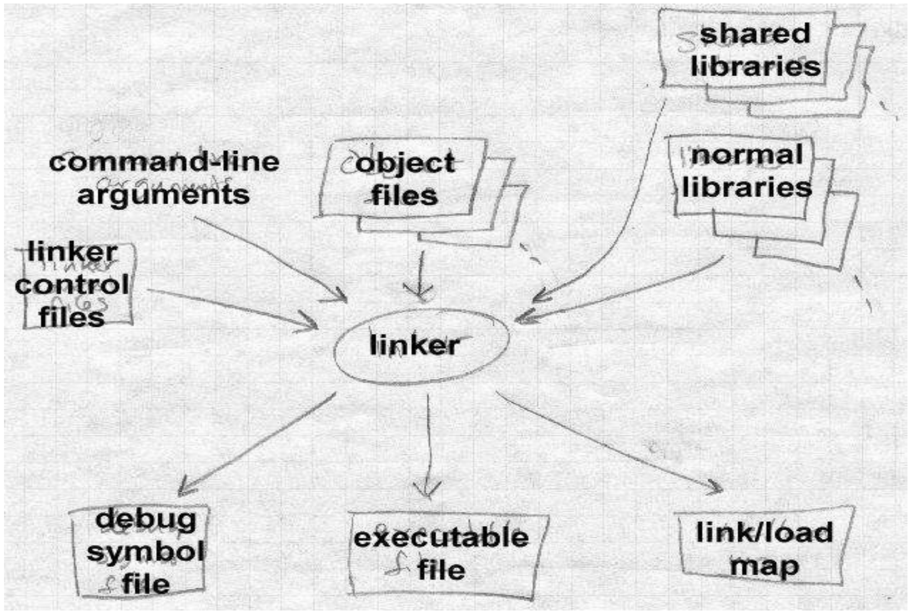
Let’s look at these two steps in more detail….

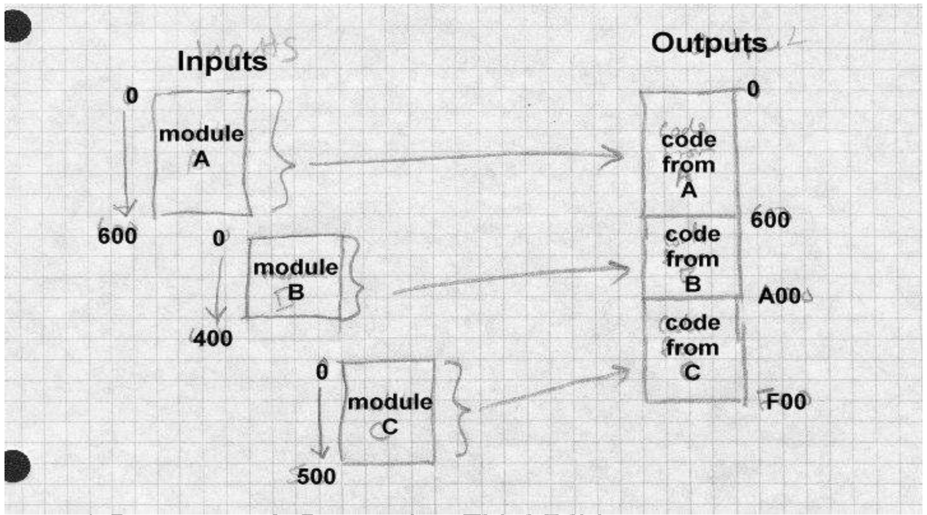
Three Kinds of Object Files (Modules)

- **可重定位目标文件(Relocatable object file) (`.o` file)**
  - 包含代码和数据，这些代码和数据可以与其他可重定位对象文件结合，形成可执行对象文件.
    - Each `.o` file is produced from exactly one source (`.c`) file
- **可执行目标文件(Executable object file) (`a.out` file)**
  - 包含可以直接复制到内存中然后执行的代码和数据.
- **可共享目标文件(Shared object file) (`.so` file)**
  - 一种特殊的可重定位对象文件，可以在load阶段或运行时动态加载到内存(either load time or run-time).
  - 在Windows中称为*Dynamic Link Libraries* (DLLs)

Review

- **Linker的主要功能**
  - 符号解析
  - 重定位
- **3类目标文件**
  - linkable
  - Executable
  - loadable
- **目标文件(object files)格式**
  - Header information
  - Object code
  - Relocation
  - Symbols
  - Debugging information
  - .....
- **ELF格式**





Executable and Linkable Format (ELF)

- **ELF 是标准的目标文件的二进制格式**
- **ELF是三类目标文件的统一格式**
  - Relocatable object files (`.o`),
  - Executable object files （`a.out`)
  - Shared object files (`.so`)

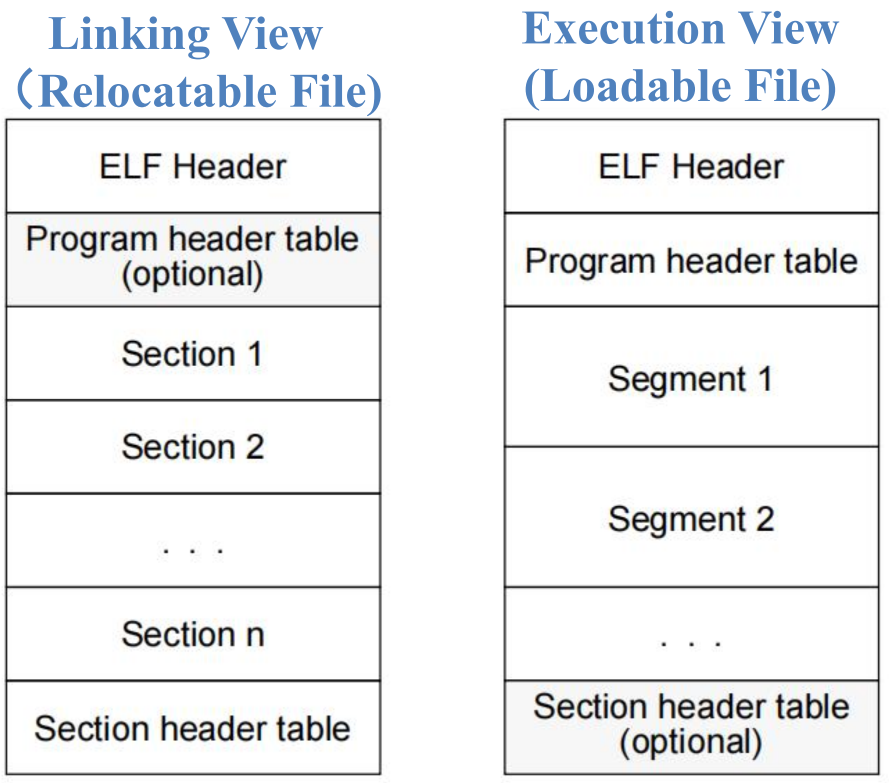
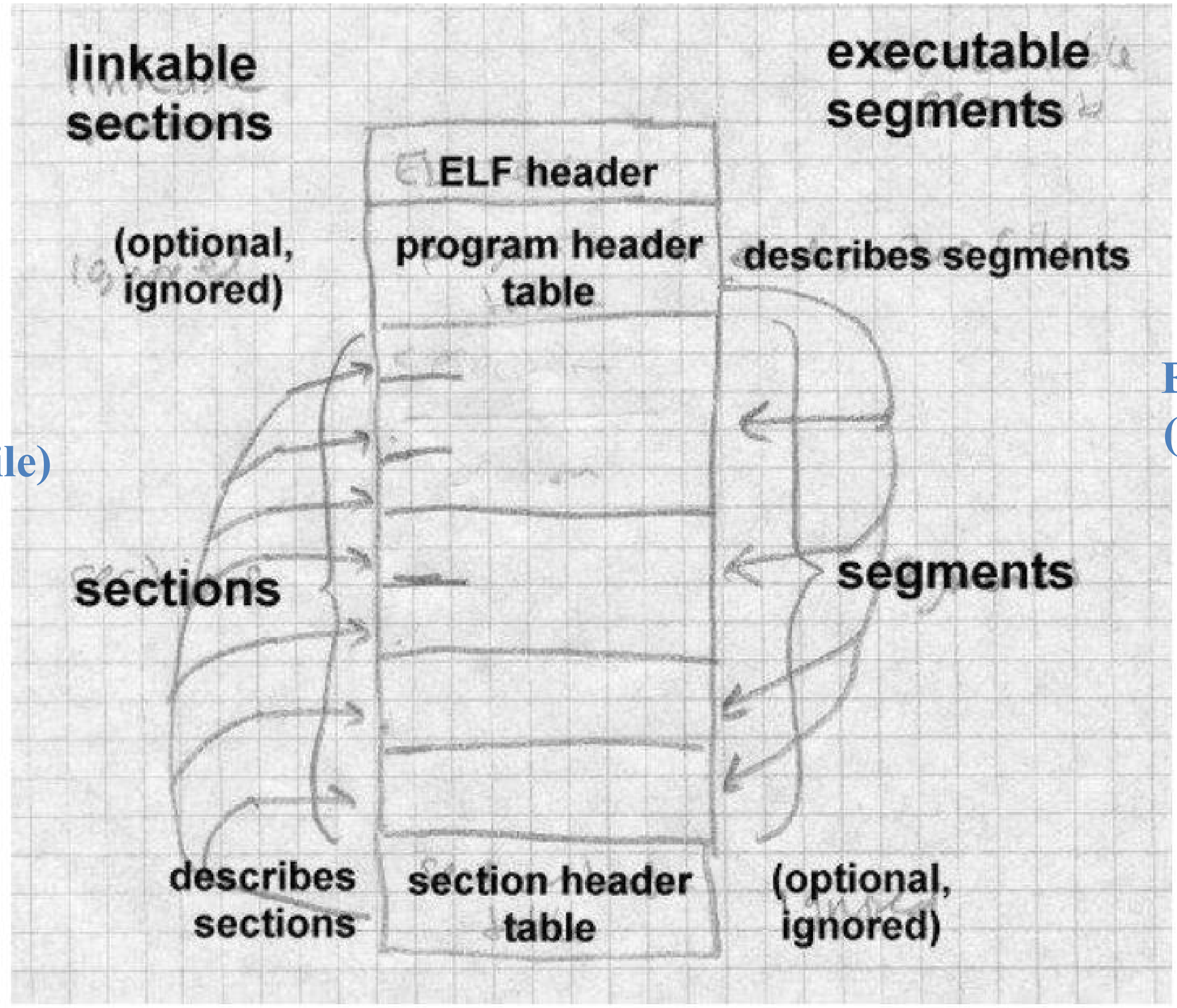


Figure 1. Structure of an ELF File

```
linux> readelf -h  /bin/ps      // 显示目标文件/bin/ps的ELF header
linux> readelf -l  /bin/ps      // 显示目标文件的Program header table
linux> readelf -S /bin/ps       // 显示目标文件的Section header table
linux> objdump -h  /bin/ps     // 显示目标文件的Section header table
linux> objdump -f /bin/ps      //显示目标文文件的file header
linux> objdump -x /bin/ps     //显示所有的headers
```

Linking View
（Relocatable File)



Execution View
(Loadable File)

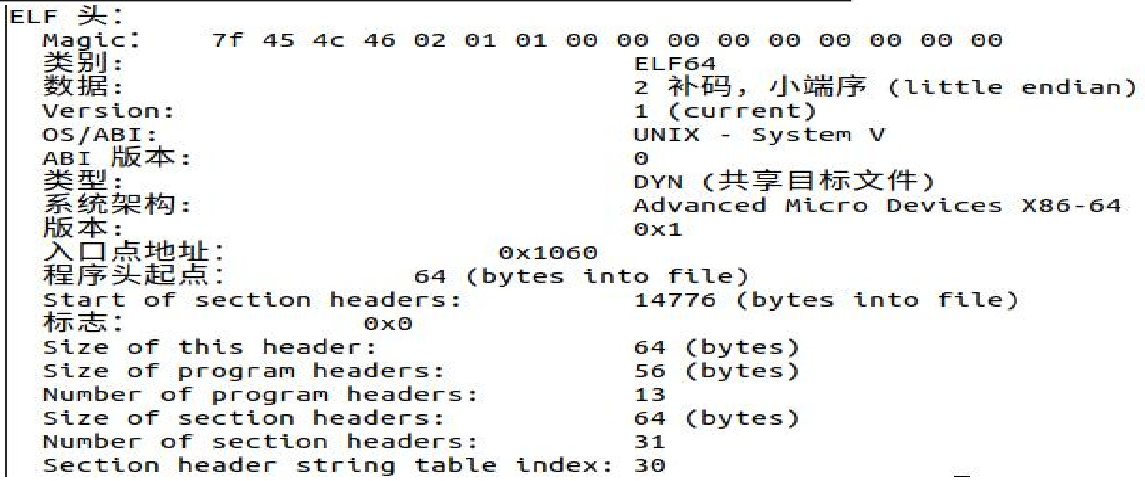
ELF Header

```
typedef struct
{
        unsigned char   e_ident[16];     /* ELF identification */
        Elf64_Half      e_type;          /* Object file type */
        Elf64_Half      e_machine;       /* Machine type */
        Elf64_Word      e_version;       /* Object file version */
        Elf64_Addr      e_entry;         /* Entry point address */
        Elf64_Off       e_phoff;         /* Program header offset *
        Elf64_Off       e_shoff;         /* Section header offset */
        Elf64_Word      e_flags;         /* Processor-specific flags
        Elf64_Half      e_ehsize;        /* ELF header size */
        Elf64_Half      e_phentsize;     /* Size of program header entry */
        Elf64_Half      e_phnum;         /* Number of program header entries */
        Elf64_Half      e_shentsize;     /* Size of section header entry */
        Elf64_Half      e_shnum;         /* Number of section header entries */
        Elf64_Half      e_shstrndx;      /* Section name string table index */
} Elf64_Ehdr;
```

Figure 2. ELF-64 Header

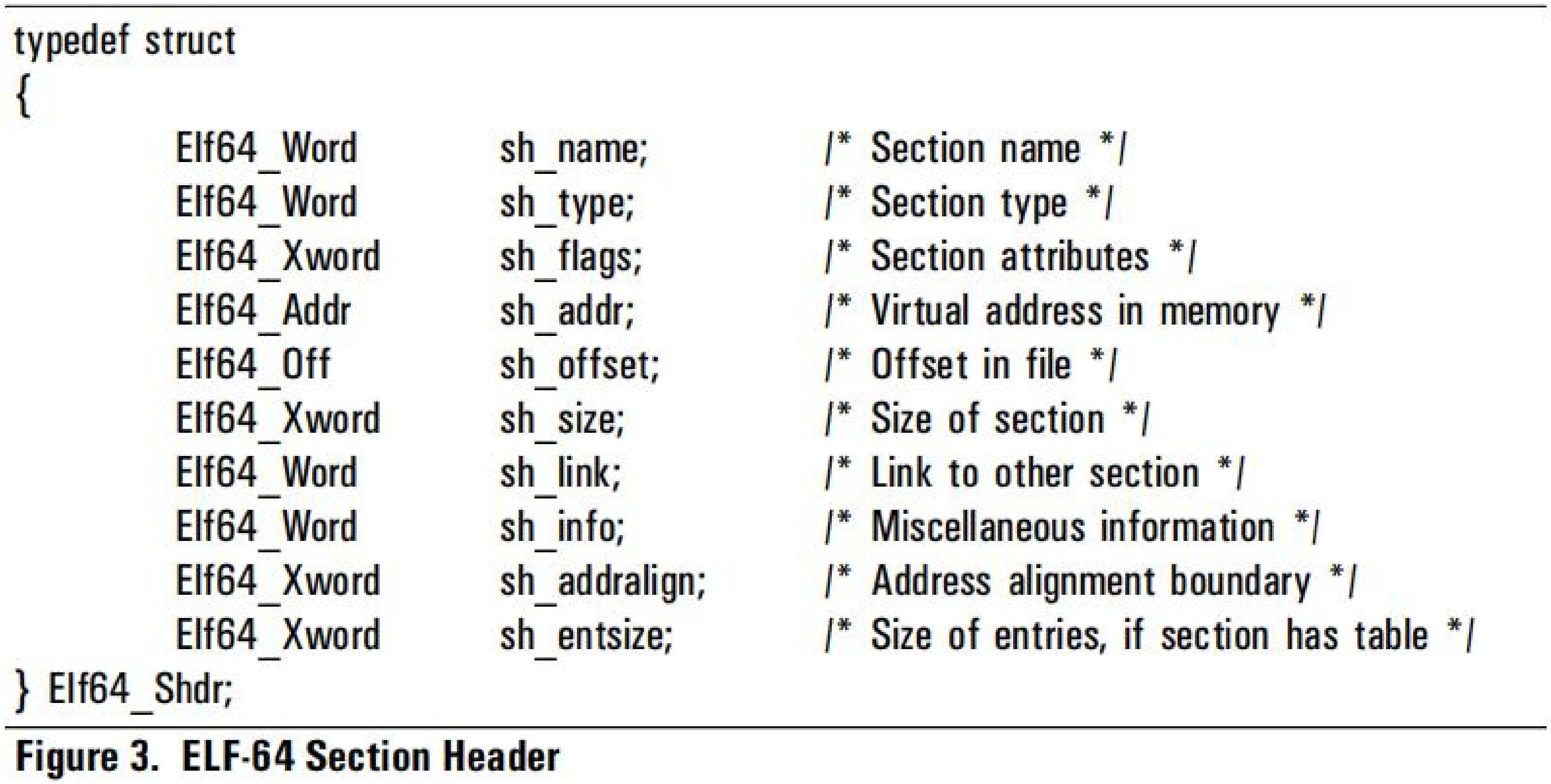
Table 1. ELF-64 Data Types

Name	*Size*	*Alignment*	*Purpose*
Elf64_Addr	8	8	Unsigned program address
Elf64_Off	8	8	Unsigned file offset
Elf64_Half	2	2	Unsigned medium integer
Elf64_Word	4	4	Unsigned integer
Elf64_Sword	4	4	Signed integer
Elf64_Xword	8	8	Unsigned long integer
Elf64_Sxword	8	8	Signed long integer
unsigned char	1	1	Unsigned small integer



```
ELF 头：
  Magic：   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF64
  数据:                              2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              DYN (共享目标文件)
  系统架构:                          Advanced Micro Devices X86-64
  版本:                              0x1
  入口点地址:                0x1060
  程序头起点:           64 (bytes into file)
  Start of section headers:          14776 (bytes into file)
  标志:                 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:         31
  Section header string table index: 30
```

Section Header



```
typedef struct
{
        Elf64_Word      sh_name;        /* Section name */
        Elf64_Word      sh_type;        /* Section type */
        Elf64_Xword     sh_flags;       /* Section attributes */
        Elf64_Addr      sh_addr;        /* Virtual address in memory */
        Elf64_Off       sh_offset;      /* Offset in file */
        Elf64_Xword     sh_size;        /* Size of section */
        Elf64_Word      sh_link;        /* Link to other section */
        Elf64_Word      sh_info;        /* Miscellaneous information */
        Elf64_Xword     sh_addralign;   /* Address alignment boundary */
        Elf64_Xword     sh_entsize;     /* Size of entries, if section has table */
} Elf64_Shdr;
```

Figure 3. ELF-64 Section Header

Section Types, sh_type

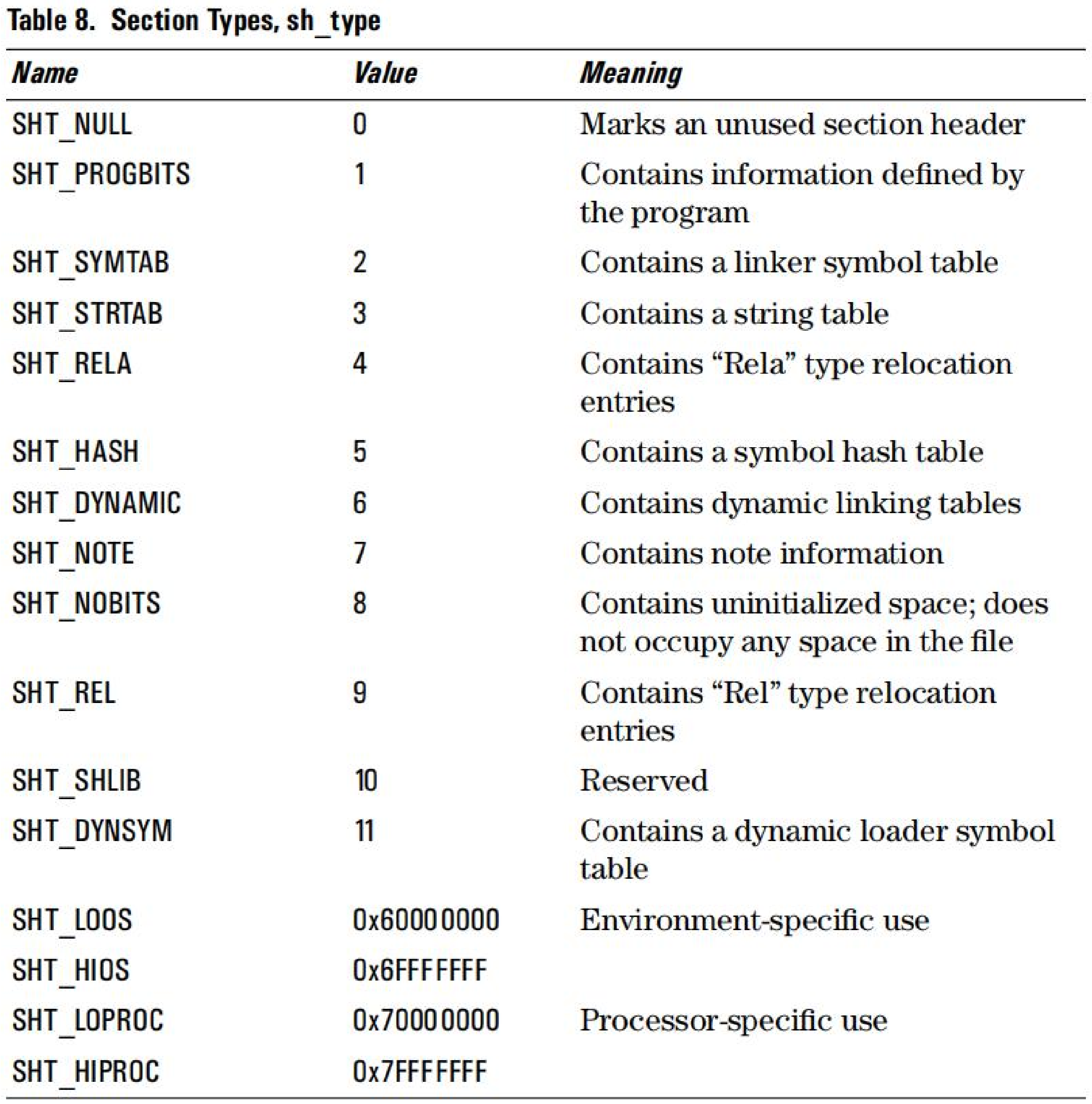


Table 8. Section Types, sh_type

Name	*Value*	*Meaning*
SHT_NULL	0	Marks an unused section header
SHT_PROGBITS	1	Contains information defined by the program
SHT_SYMTAB	2	Contains a linker symbol table
SHT_STRTAB	3	Contains a string table
SHT_RELA	4	Contains "Rela" type relocation entries
SHT_HASH	5	Contains a symbol hash table
SHT_DYNAMIC	6	Contains dynamic linking tables
SHT_NOTE	7	Contains note information
SHT_NOBITS	8	Contains uninitialized space; does not occupy any space in the file
SHT_REL	9	Contains "Rel" type relocation entries
SHT_SHLIB	10	Reserved
SHT_DYNSYM	11	Contains a dynamic loader symbol table
SHT_LOOS	0x60000000	Environment-specific use
SHT_HIOS	0x6FFFFFFF	
SHT_LOPROC	0x70000000	Processor-specific use
SHT_HIPROC	0x7FFFFFFF	

- **PROGBITS**
  - 包含code, data, debugger 信息
- **NOBITS**
  - 包含不需初始化的数据，在文件中不占空间，但在程序装载时要分配内存空间（for BSS)
- **SYMTAB**
- **STRTAB**
- **REL and RELA**
- **DYNAMIC and HASH**
- **.....**

Section Attributes, sh_flags

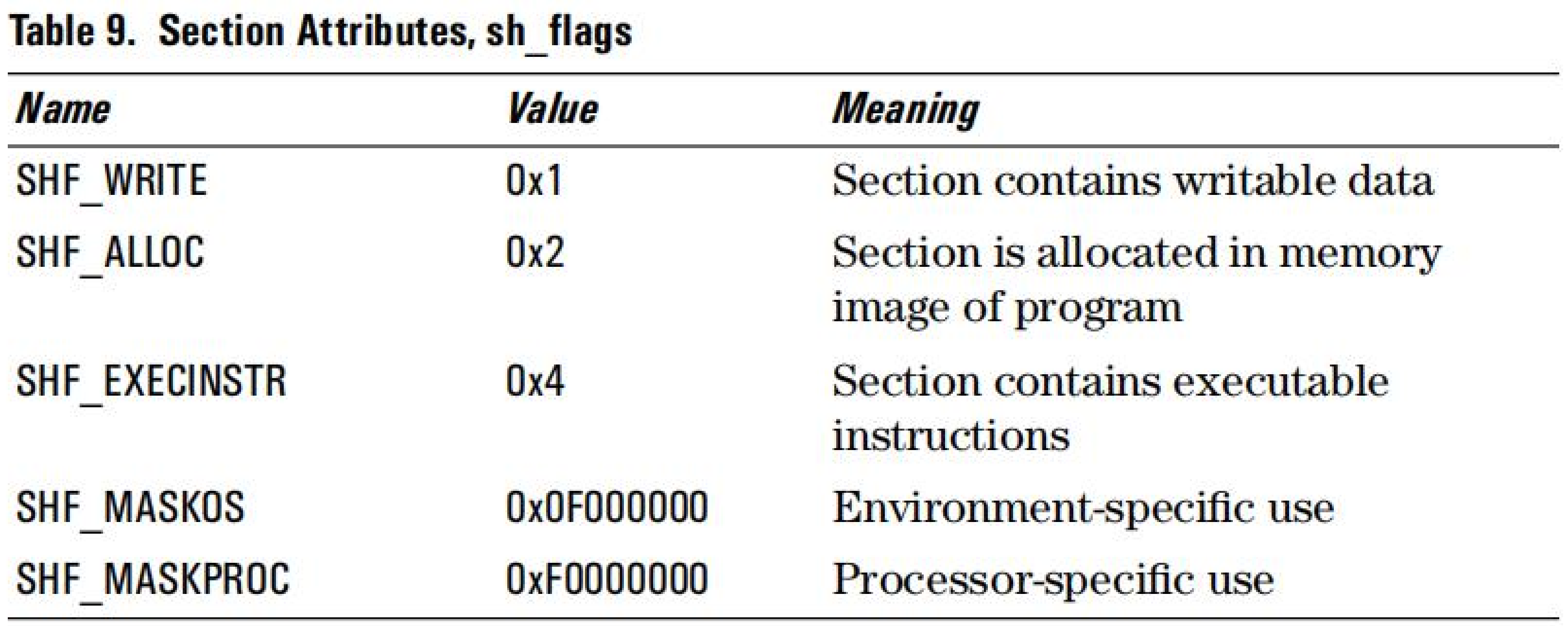
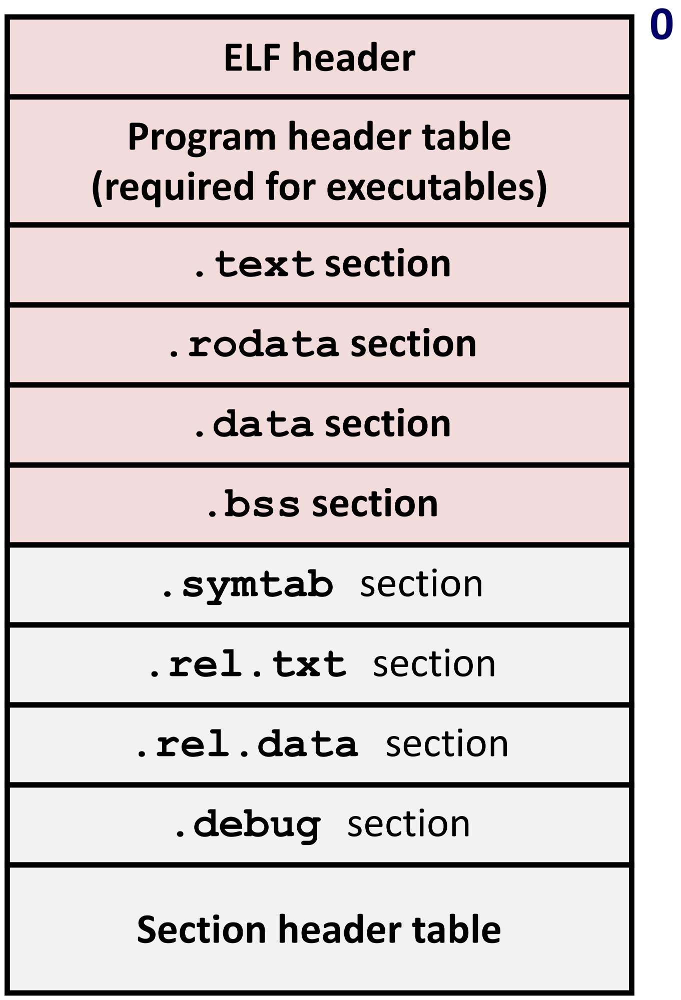


Table 9. Section Attributes, sh_flags

Name	*Value*	*Meaning*
SHF_WRITE	0x1	Section contains writable data
SHF_ALLOC	0x2	Section is allocated in memory image of program
SHF_EXECINSTR	0x4	Section contains executable instructions
SHF_MASKOS	0x0F000000	Environment-specific use
SHF_MASKPROC	0xF0000000	Processor-specific use

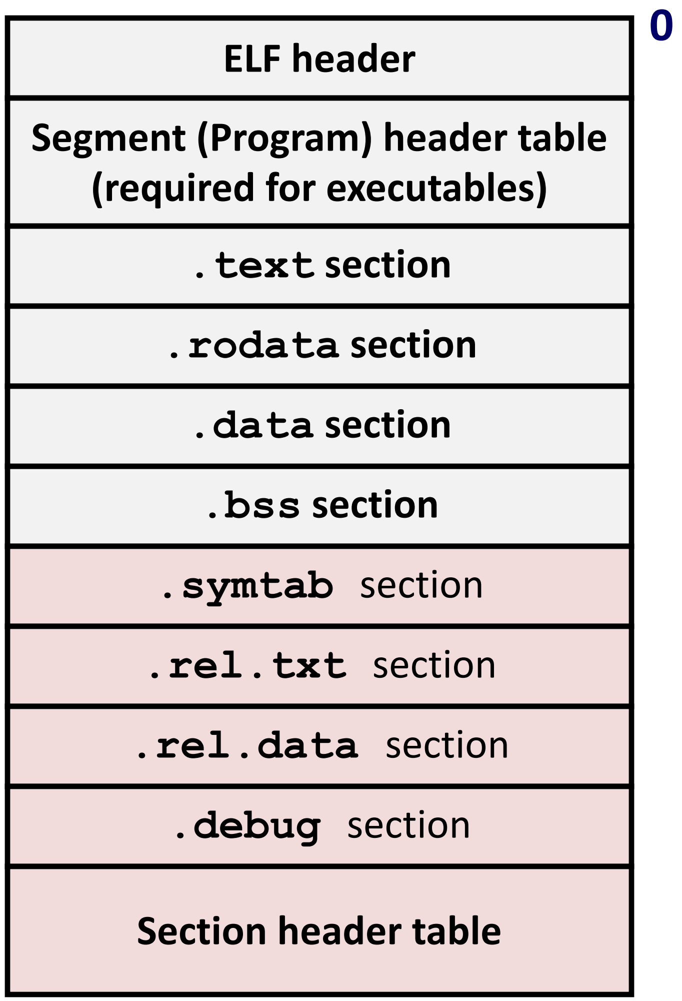
ELF Object File Format

- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Program header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **`.text` section**
  - Code
  - PROGBITS， ALLOC+EXECINSTR
- **`.rodata` section**
  - Read only data: jump tables, ...
  - PROGBITS， ALLOC
- **`.data` section**
  - Initialized global variables
  - PROGBITS， ALLOC+WRITE
- **`.bss` section （NOBITS＋ALLOC）**
  - 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



Outline

- **链接技术核心基础**
  - 链接的动机与核心价值
  - 目标文件的分类
  - 链接器的核心功能
- **符号解析与重定位**
- **代码重用**
  - 静态链接库
  - 动态链接库
- **链接器的应用**
  - 程序的模块化构建
  - 符号控制与冲突消解
  - 代码复用
  - 调试与监控：库插桩技术

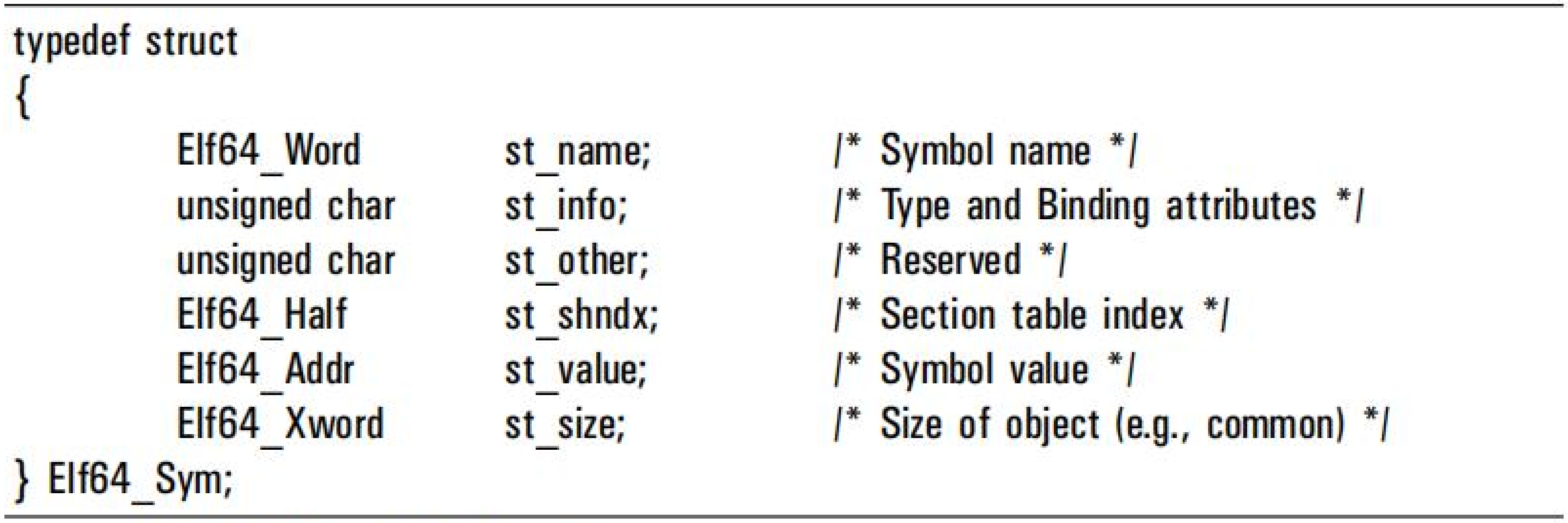
Linker Symbols

- **全局符号(Global symbols)-由模块m定义可被其他模块引用**
  - 模块m定义的符号，其他模块可以引用.
  - E.g.: non-`static` C functions and non-`static` global variables.

- **外部符号(External symbols)-由其他模块定义被模块m引用**
  - 模块m引用的全局符号，但由其他模块定义.

- **本地符号(Local symbols)-被模块m定义和引用**
  - 仅由模块m定义和引用的符号.
  - E.g.: C functions and global variables defined with the `static` attribute.
  - 本地链接符号不是指程序的局部变量 （**不是程序中的局部变量)**

- **显示符号表内容**
  - eg. `linux>`**`readelf -s`**` test.o` or **`objdump -t`**` test.o`



```
typedef struct
{
        Elf64_Word        st_name;        /* Symbol name */
        unsigned char     st_info;        /* Type and Binding attributes */
        unsigned char     st_other;       /* Reserved */
        Elf64_Half        st_shndx;       /* Section table index */
        Elf64_Addr        st_value;       /* Symbol value */
        Elf64_Xword       st_size;        /* Size of object (e.g., common) */
} Elf64_Sym;
```

Figure 4. ELF-64 Symbol Table Entry

Symbol Bindings
Symbol Types

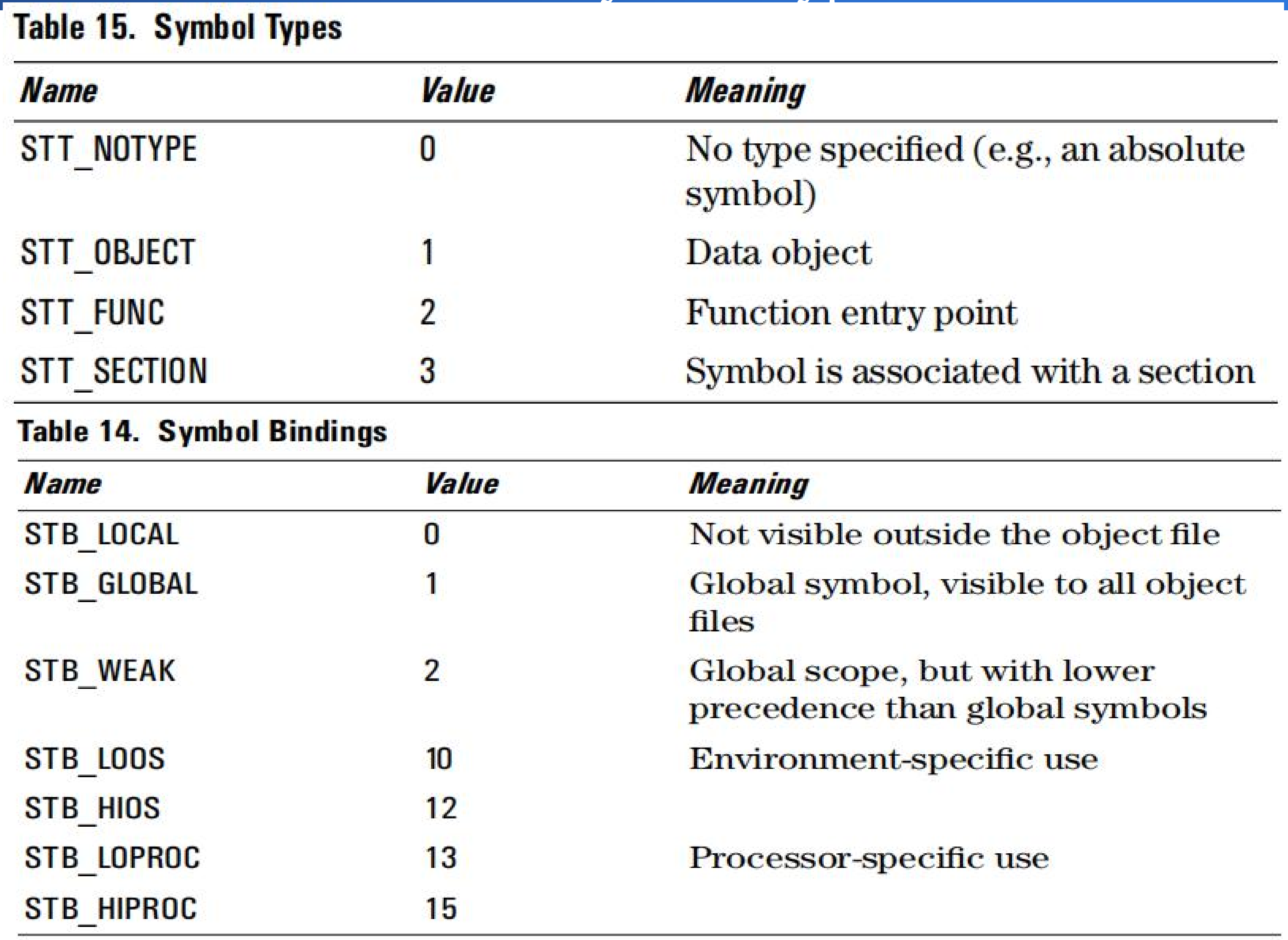


Table 15. Symbol Types

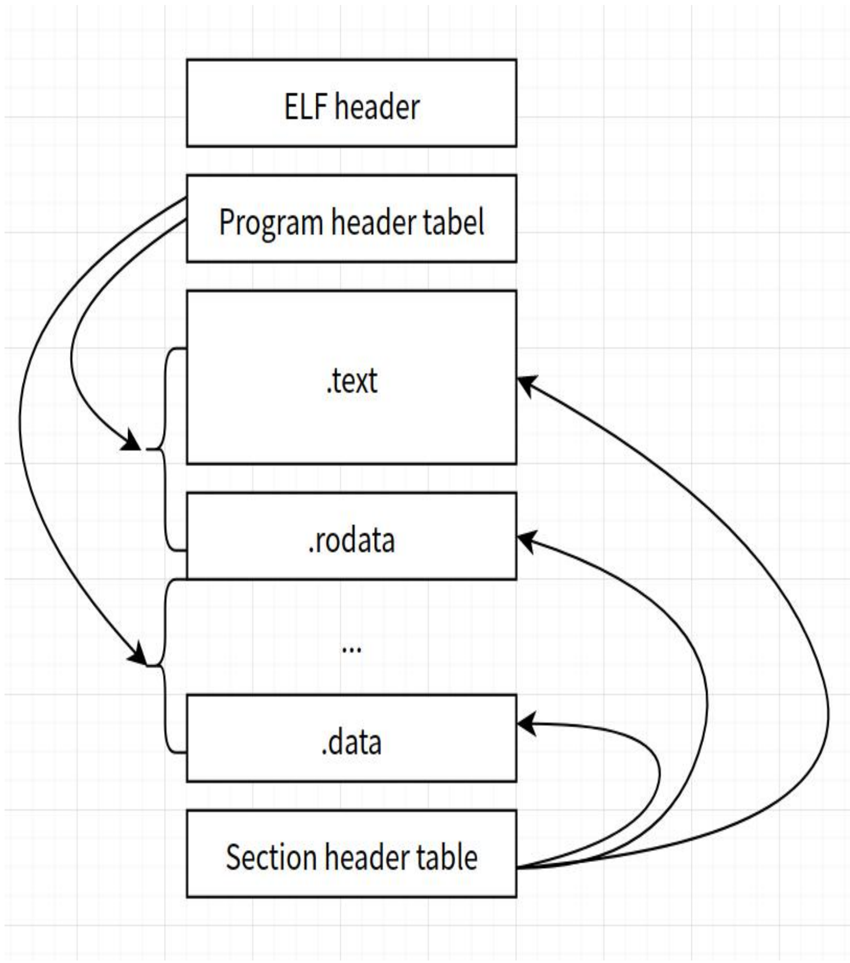
Name	*Value*	*Meaning*
STT_NOTYPE	0	No type specified (e.g., an absolute symbol)
STT_OBJECT	1	Data object
STT_FUNC	2	Function entry point
STT_SECTION	3	Symbol is associated with a section

Table 14. Symbol Bindings

Name	*Value*	*Meaning*
STB_LOCAL	0	Not visible outside the object file
STB_GLOBAL	1	Global symbol, visible to all object files
STB_WEAK	2	Global scope, but with lower precedence than global symbols
STB_LOOS	10	Environment-specific use
STB_HIOS	12	
STB_LOPROC	13	Processor-specific use
STB_HIPROC	15	

Review

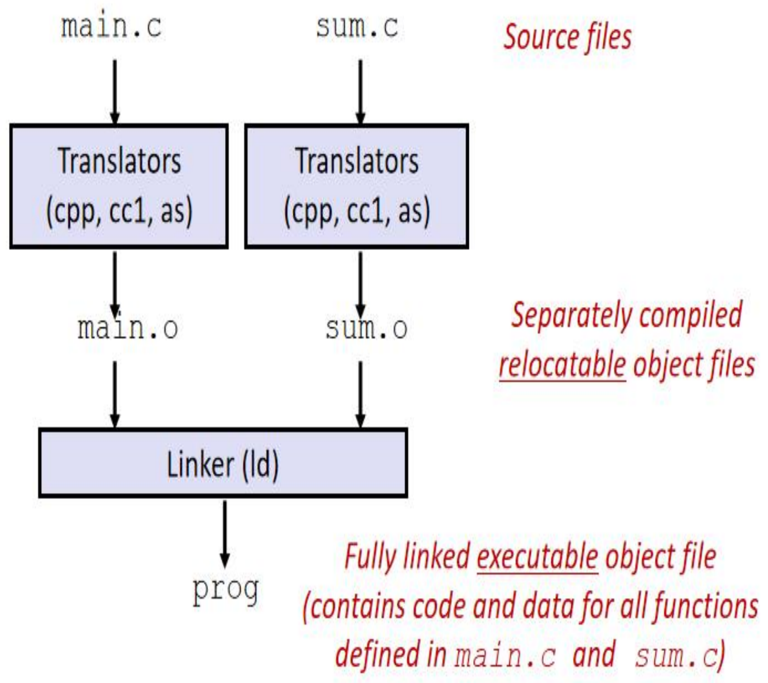
- **为什么需要linker?**
  - 模块化开发，高效构建，代码复用
- **Linker的主要功能：**
  - 将多个目标文件中代码和数据片段收集并组合成为一个单一文件的过程。包括：
    - **符号解析**
    - **重定位**
- **Linker的应用场景**
  - 日常开发：编译多文件程序
  - 系统级优化：动态库共享内存
  - 调试与监控：库插桩追踪函数调用
- **3类目标文件**
  - linkable，Executable，loadable
- **ELF格式**
  - ELF 文件头信息
  - 段(Segment)头表和段信息 (执行视图)
  - 节(sections)头表和节 ：code, data, relocation, symbol, debugging information etc.



静态链接与重定位原理

- **静态链接的核心流程**
  - 定义：编译时将所有依赖的.o 文件和静态库（.a）合并为单一可执行文件
  - 两步扫描流程：
    - Pass1：读取所有.o 文件的节大小，计算内存布局，构建全局符号表
    - Pass2：合并节数据，执行重定位，更新符号引用地址
- **符号解析与冲突处理**
- **重定位原理与实现**

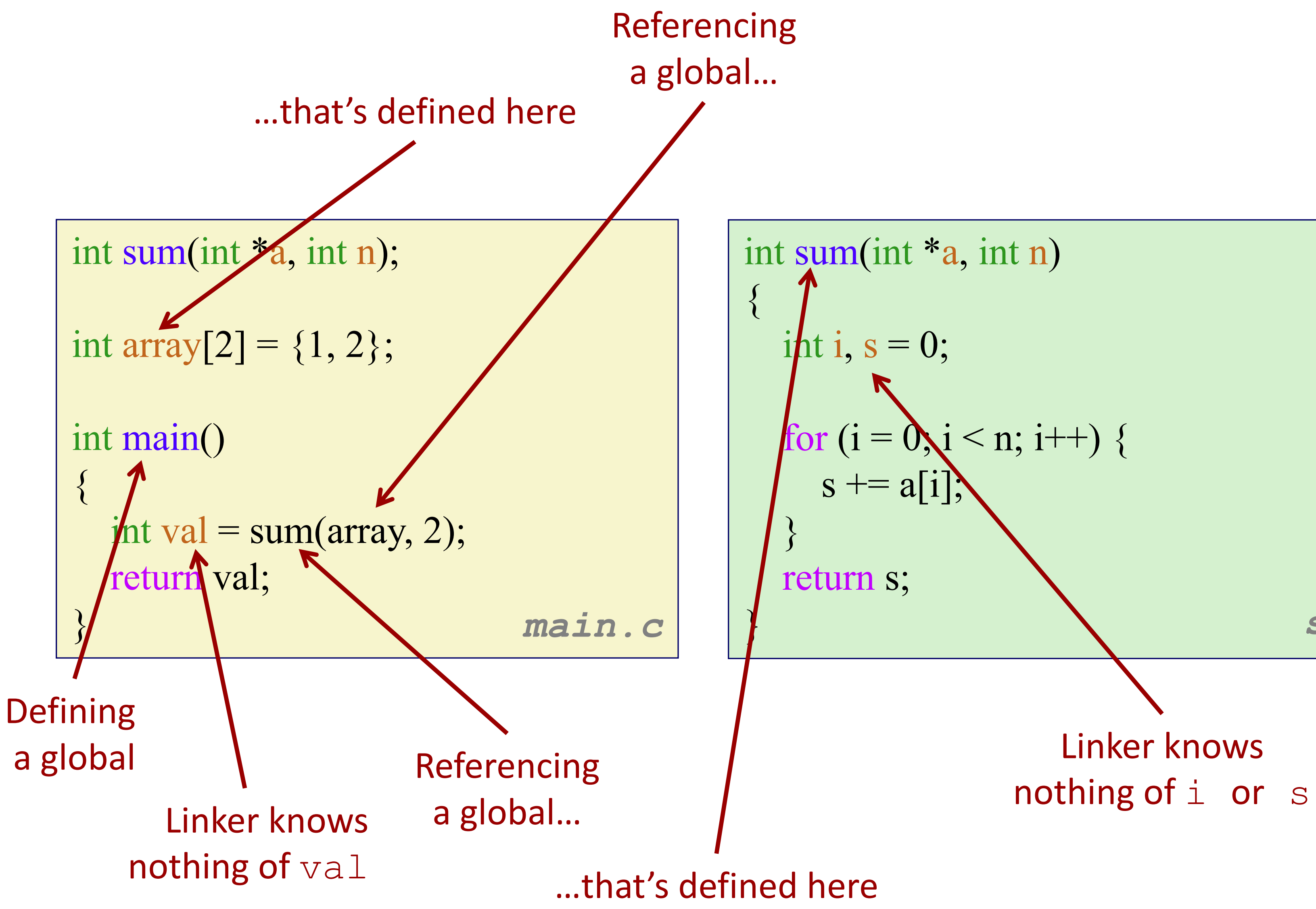
- **程序使用编译器编译和链接:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`


main.c
sum.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)

符号解析：Linker Symbols

- **符号解析**
  - 将程序中所有符号引用与唯一的符号定义建立一一对应关系，消除符号歧义，为后续地址重定位提供准确的符号定位基础
- **全局符号(Global symbols)-由模块m定义可被其他模块引用**
  - 模块m定义的符号，其他模块可以引用.
  - E.g.: non-`static` C functions and non-`static` global variables.
- **外部符号(External symbols)-由其他模块定义被模块m引用**
  - 模块m引用的全局符号，但由其他模块定义.
- **本地符号(Local symbols)-只被模块m定义和引用**
  - 仅由模块m定义和引用的符号.
  - E.g.: C functions and global variables defined with the `static` attribute.
  - 本地链接符号不是指程序的局部变量 （**不是程序中的局部变量)**
- **显示符号表内容**
  - eg. `linux>`**`readelf -s`**` test.o` or **`objdump -t`**` test.o`

Step 1: Symbol Resolution



```
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

Symbol Identification

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

symbols.c:

```
int incr = 1;
extern int g;
int g = 0;
static int foo(int a) {
   int b = a + incr;
   return b;
}
int main(int argc,
         char* argv[]) {
   printf("%d\n", foo(5));
   return 0;
}
```

Names:

- **incr**
- **foo**
- **a**
- **argc**
- **argv**
- **b**
- **main**
- **printf**
- **“%d\n”**
- **g**

Can find this with readelf:

```
linux> readelf -s symbols.o
```

Local Symbols

- **局部非静态C变量 vs. 局部静态变量**
  - local non-static C variables: stored on the stack
  - local static C variables: stored in either `.bss,` or `.data`

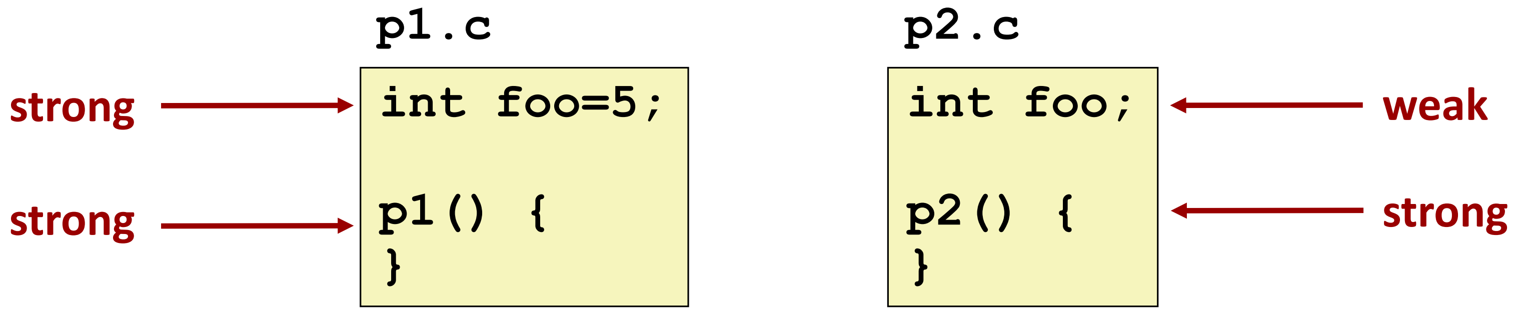
```
int f()
{
    static int x = 0;
    return x;
}

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

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

Creates local symbols in the symbol table with unique names, e.g., `x.1` and `x.2`.

How Linker Resolves Duplicate Symbol Definitions

- **程序中的符号分为强符号和弱符号(*strong* or *weak*)**
  - *Strong*: 函数名和初始化的全局变量名
  - *Weak*: 未初始化的全局变量名或用_attribute_((weak))显示声名的函数名
    - Or ones declared with specifier **`extern`**



p1.c

```
int foo=5;

p1() {
}
```

p2.c

```
int foo;

p2() {
}
```

Linker's Symbol Rules

- **Rule 1: 不允许多个同名的强符号定义**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: 如果有多个弱符号和一个强符号定义，则选择强符号定义**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: 如果仅有多个同名的弱符号**
  - 若是弱变量（空间优先）
    - gcc的linker选择占用内存空间最大的弱变量
  - 若是弱函数（顺序优先）
    - gcc的linker选择第一个在链接命令中出现的弱函数
- `gcc -fno-common`**选项:** 强制将未初始化全局变量视为 “强符号”

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!

Important: Linker does not do type checking.

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

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

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

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Link example

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

```
/* bar1.c*/
int main ()
{
  return 0;
}
```

Linker生成错误信息，强符号main被定义两次

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

```
/* bar2.c*/
int x = 15213;
void f ()
{
}
```

Linker生成错误信息，强符号x被定义两次

Link example

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

```
/* bar3.c*/
int x;

void f ()
{
  x=15212;
}
```

Linker不报错，选择强符号定义foo3.c中的x定义。但是打印结果是15212

Link example

```
/* foo4.c*/
#include <stdio.h>
void f(void);
int x;
int main()
{
   x=15213;
   f();
   printf("x=%d\n",x);
   return 0;
}
```

```
/* bar4.c*/
int x

void f()
{
   x=15212;
}
```

Linker不报错，任意选择x的定义。但是打印结果是15212

TIPS： 在-fno-common选项时报错

Link example

```
/* foo5.c*/
#include <stdio.h>
void f(void);
int x=15213;
int y=15212;
int main ()
{
  x=15213;
  f();
  printf(“x=0x%x y= 0x%x\n”,x,y);
  return 0;
}
```

```
/* bar5.c*/
double x;

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

TIPS：在-fno-common选项时报错

- linker不报错，选择foo5.c中x的定义，但是打印结果可能不是预想的值。f()中对x的赋值会覆盖y的值。

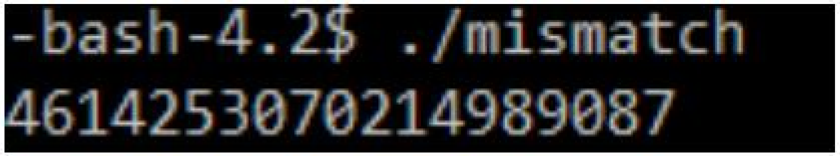
```
linux> gcc -Wall -Og -o foobar5 foo5.c bar5.c
/usr/bin/ld: Warning: alignment 4 of symbol `x' in /tmp/cclUFK5g.o is smaller
than 8 in /tmp/ccbTLcb9.o
linux> ./foobar5
 x = 0x0 y = 0x80000000
```

Type Mismatch Example

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

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

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

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



```
-bash-4.2$ ./mismatch
4614253070214989087
```

Detecting the Type Mismatch Example

```
long int x;

                                mismatch.h
```

```
#include "mismatch.h"

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

```
#include "mismatch.h"
double x = 3.14;
```

- **Now we get an error … from the compiler, not the linker.**
  - mismatch-variable.c:3:8: conflicting types for 'x'
  - mismatch.h:1:17: previous declaration of 'x'

Rules for avoiding type mismatches

- **尽可能避免使用全局变量**
- **在.h头文件中声明所有非静态变量或函数**
  - Make sure to include the header file everywhere it's relevant
  - Including the files that define those symbols
- **将extern全局符号声明放在头文件的声明中**
  - Unnecessary but harmless for function declarations
  - Avoids the quirky behavior of extern-less global variables
- **其他提示**
  - 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

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)

c1.c

```
#include "global.h"

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

global.h

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

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

c2.c

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

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;
}
```

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

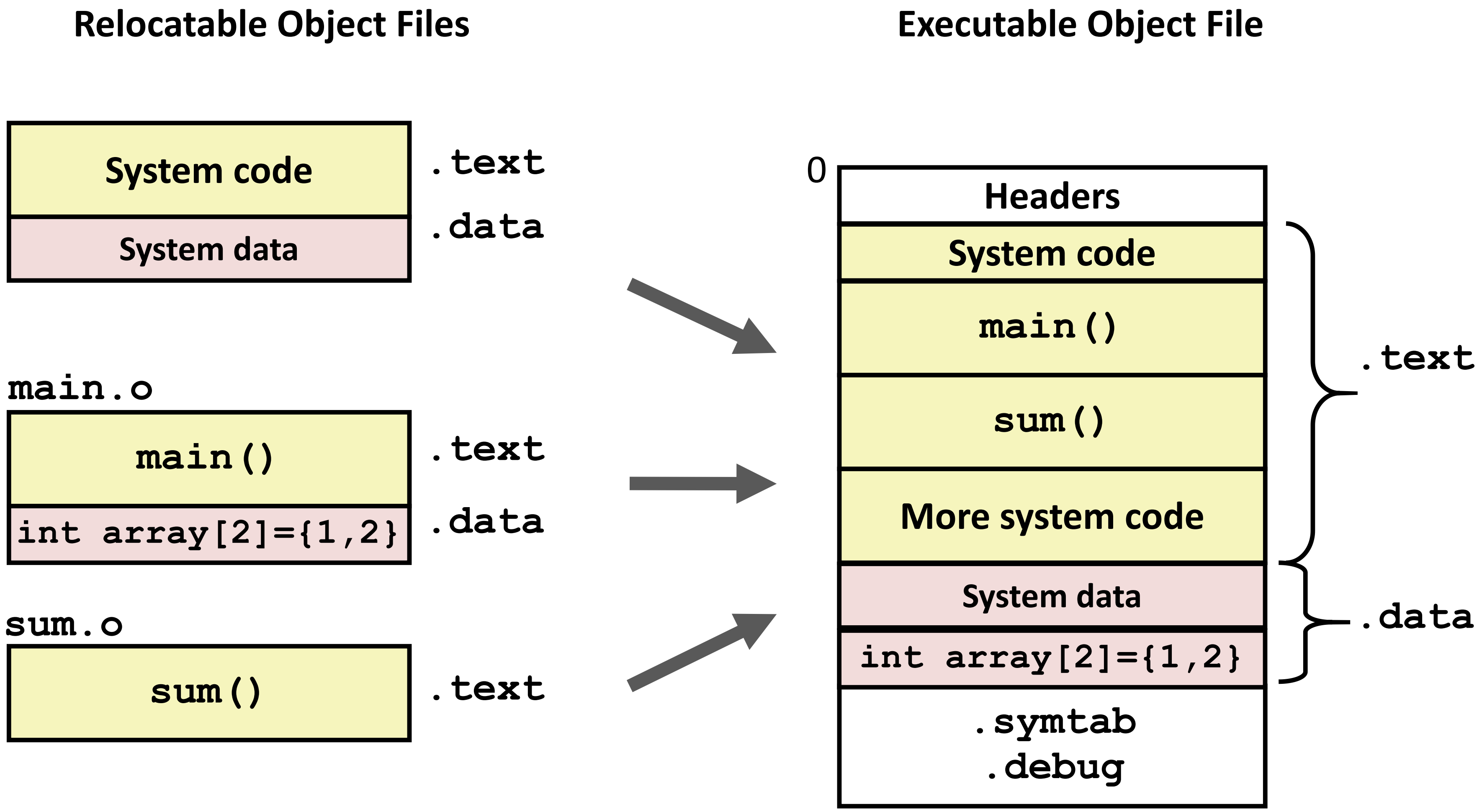
What Do Linkers Do? (cont’ d)

- **Step 2: Relocation**
  - 将各模块的代码和数据部分按类型合并。
  - 将symbols在.o文件中的相对位置**重新定位**到可执行文件中它们的最终绝对内存(相对)位置
  - 用这些符号的新位置来更新对这些符号的所有引用
- **Linking Example**

```
int sum(int *a, int n);
int array[2] = {1, 2};
int main(int argc,char **argv)
{
    int val = sum(array, 2);
    return val;
}
```

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

Step 2: Relocation



Linker executes in two passes:

Pass 1: 读取节(section)大小，计算最终内存布局。同时读取所有符号，在内存中创建完整符号表

Pass 2: 读取节和重定位信息，更新地址，写入新文件.

Relocation Entries

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

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

```
linux>objdump -r main.o
linux>readelf -r main.o
        //显示main.o中的需重定位项
```

```
o'p'b<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
```

Using PC-relative addressing for sum(): 0x4004e8 = 0x4004e3 + 0x5

Source: `objdump -dx prog`

Another Example C Program linking in X86-32

main.c

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

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

swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

Symbol tables

- **符号表是由汇编器使用编译器导出到.s文件中的符号来构建的.**
- **ELF符号表包含在.symtab section中。每个符号占有一个entry，每个entry的内容如下：**

```
typedef struct {
   Elf32_Word   st_name;   /*string table offset*/
   Elf32_Addr    st_value;   /*section offset, or VM address*/
   Elf32_Word    st_size;     /*object size in bytes*/
   unsigned char type:4,      /*data, func, section, or src file name*/
                  binding:4; /*local or global*/
   unsigned char reserved;  /*unused*/
   Elf32_Half    st_shndx ; /*section header index, ABS,UNDEF,*/
                              /*or COMMON*/
} Elf32_Sym;
```

```
typedef struct {
   Elf64_Word   st_name;
   Elf64_Addr   st_value;
   Elf64_Word   st_size;
   unsigned char type:4,
                  binding:4;
   unsigned char reserved;
   Elf64_Half    st_shndx ;

} Elf64_Sym;
```

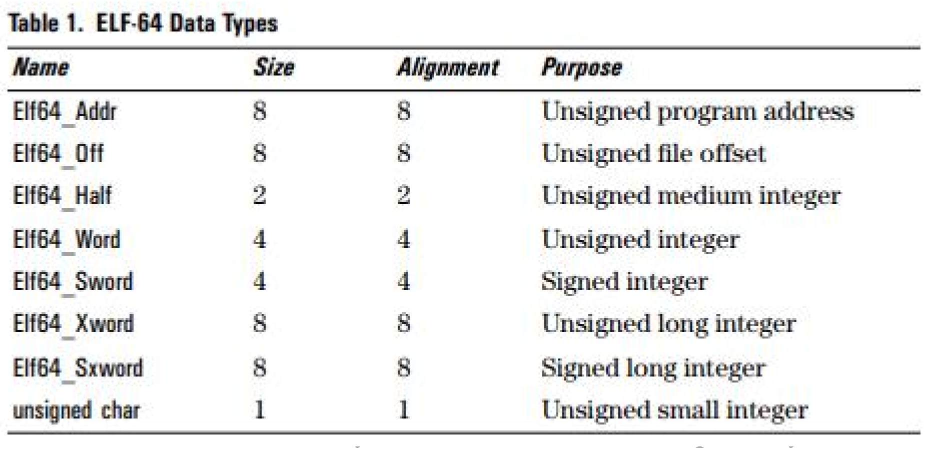
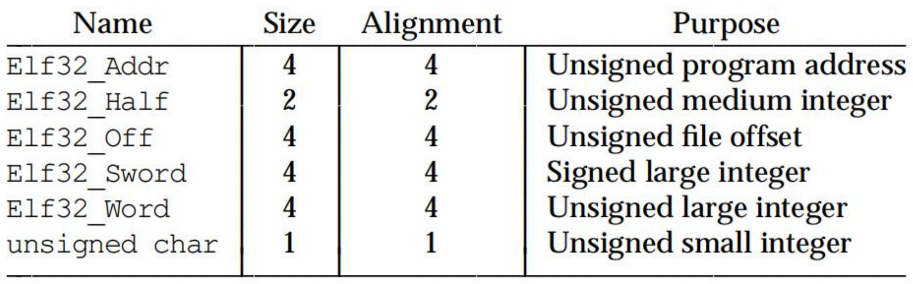


Table 1. ELF-64 Data Types

Name	*Size*	*Alignment*	*Purpose*
Elf64_Addr	8	8	Unsigned program address
Elf64_Off	8	8	Unsigned file offset
Elf64_Half	2	2	Unsigned medium integer
Elf64_Word	4	4	Unsigned integer
Elf64_Sword	4	4	Signed integer
Elf64_Xword	8	8	Unsigned long integer
Elf64_Sxword	8	8	Signed long integer
unsigned char	1	1	Unsigned small integer



Name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

Symbol tables

- **main.o: the last three GLOBAL symbols, the first eight are local symbols that linker uses internally**

Num:	Value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	OBJECT	GLOBAL	0	3	buf
9:	0	17	FUNC	GLOBAL	0	1	main
10:	0	0	NOTYPE	GLOBAL	0	UND	swap

  - .Ndx=1 .text section ; Ndx=3 .data section

- **swap.o**

Num:	Value	Size	Type	Bind	Ot	Ndx	Name
8:	0	4	OBJECT	GLOBAL	0	3	bufp0
9:	0	0	NOTYPE	GLOBAL	0	UND	buf
10:	0	39	FUNC	GLOBAL	0	1	swap
11:	4	4	OBJECT	GLOBAL	0	COM	bufp1

  - 最后的条目是全局符号bufp1，一个4字节未初始化的数据对象（具有4字节对齐要求），最终将在链接此模块时分配为.bss对象

Relocation entry (ELF32)

两种格式：`Elf32_Rel`紧凑格式 addend隐含地在需定位的字中，`Elf32_Rela`格式显示给出addend

```
typedef struct {
   ELf32_Addr offset; /*offset of the reference to relocate*/
   Elf32_Word symbol:24,/*symbol the reference should point to*/
              type:8;   /*relocation type*/
} Elf32_Rel;


typedef struct {
    ELf32_Addr offset;    /*offset of the reference to relocate*/
    Elf32_Word symbol:24, /*symbol the reference should point to*/
          type:8;         /*relocation type*/
    Elf32_Sword addend;   /*Constant part of relocation expression*/
} Elf32_Rela;
```

R_386_PC32: PC relative address relocation

R_386_32: Absolute address relocation

两种基本的重定位类型：
1) 使用32位PC相对地址的引用
2) 使用32位绝对地址的引用

Relocation entry (ELF64)

两种格式：

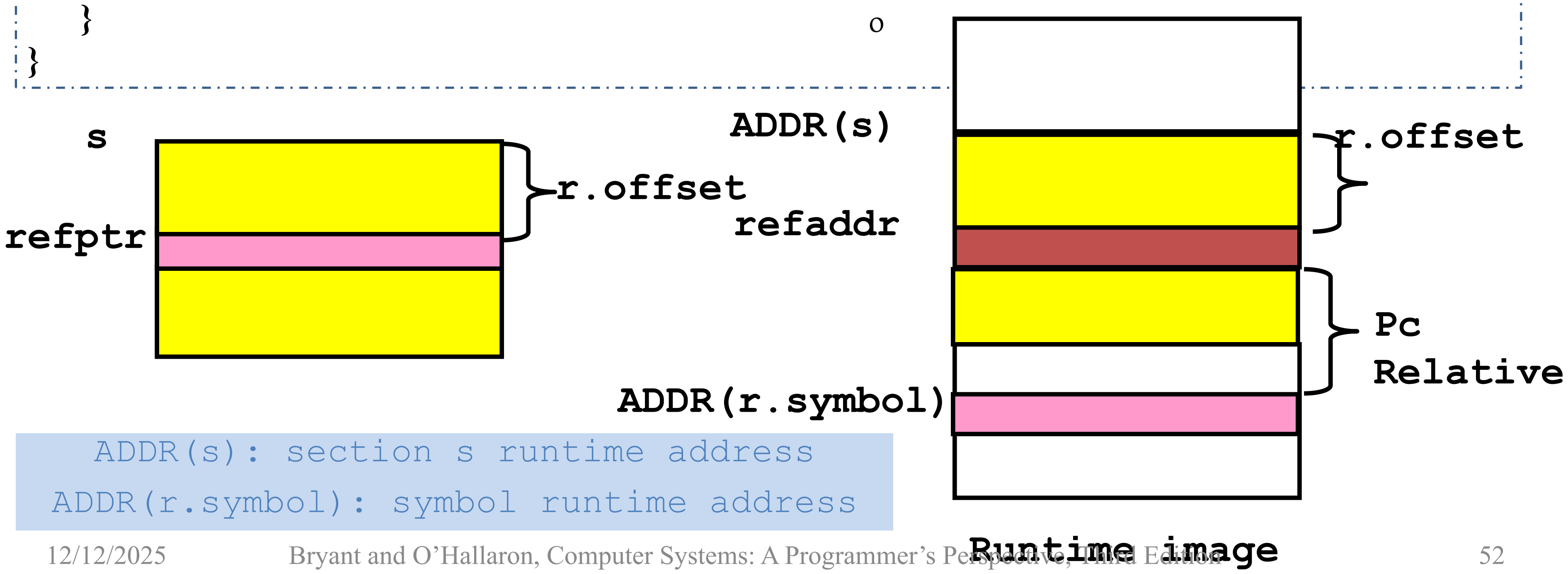
```
typedef struct {
    Elf64_Addr offset; /*offset of the reference to relocate*/
    Elf64_Xword  type:32, /*relocation type*/
                 symbol:32;
                /*symbol table index the reference should point to*/
} Elf64_Rel;
```

```
typedef struct {
    Elf64_Addr  offset;  /*offset of the reference to relocate*/
    Elf64_Xword  type:32, /*relocation type*/
          symbol:32;
       /*symbol table index the reference should point to*/
    Elf64_Sword addend;  /*Constant part of relocation expression*/
} Elf64_Rela;
```

R_X86_64_PC32; R_X86_64_32 /X86-64 small code model, 2GB address space

Relocation Algorithm

```
foreach section s {
    foreach relocation entry r {
        refptr = s + r.offset;                    /* ptr to reference to be relocated*/
        /* relocate  a pc-relative reference*/
        if ( r.type == R_386_PC32) {    /*   if(r.type==R_X86_64_PC32) */
            refaddr = ADDR(s) + r.offset;               /*ref's run-time address*/
            *refptr = (unsigned)(ADDR(r.symbol) + r.addend – refaddr);
        }
        if ( r.type == R_386_32)   *refptr = (unsigned)(ADDR(r.symbol) + r.addend );
        /* if (r.type == R_X86_64_32) */
    }
}
```



ADDR(s): section s runtime address

ADDR(r.symbol): symbol runtime address

main.o Relocation Info

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

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

```
0000000 <main>:
   0:    55                    push    %ebp
   1:    89 e5                 mov     %esp,%ebp
   3:    83 ec 08              sub     $0x8,%esp
   6:    e8 fc ff ff ff        call    7 <main+0x7>
                               7: R_386_PC32 swap
   b:    31 c0                 xor     %eax,%eax
   d:    89 ec                 mov     %ebp,%esp
   f:    5d                    pop     %ebp
  10:    c3                    ret
```

```
Disassembly of section .data:

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

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

swap.o Relocation Info (.text)

```
extern int buf[];

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

void swap()
{
  int temp;

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

raddend

```
Disassembly of section .text:

00000000 <swap>:
 0: 55                       push   %ebp
 1: 8b 15 00 00 00 00        mov    0x0,%edx
                             3: R_386_32 bufp0
 7: a1 04 00 00 00            mov    0x4,%eax
                             8: R_386_32 buf
 c: 89 e5                    mov    %esp,%ebp
 e: c7 05 00 00 00 00 04movl   $0x4,0x0
15: 00 00 00
                             10: R_386_32 bufp1
                             14: R_386_32 buf
18: 89 ec                    mov    %ebp,%esp
1a: 8b 0a                    mov    (%edx),%ecx
1c: 89 02                    mov    %eax,(%edx)
1e: a1 00 00 00 00           mov    0x0,%eax
                             1f: R_386_32 bufp1
23: 89 08                    mov    %ecx,(%eax)
25: 5d                       pop    %ebp
26: c3                       ret
```

swap.o Relocation Info (.data)

```
extern int buf[];

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

void swap()
{
  int temp;

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

```
Disassembly of section .data:

00000000 <bufp0>:
   0:    00 00 00 00

         0: R_386_32 buf
```

Relocate PC-relative reference

6: e8 fc ff ff ff call 7 <main+0x7>

7: R_386_PC32 swap -4

r.offset = 0x7

r.symbol = swap

r.type = R_386_PC32

refaddr: 是要重写位置的首地址。refaddr + abs(raddend)是 call指令的下一条地址。

合并后swap地址为080483c8

ADDR(s)= ADDR(.text) = 0x80483b4

ADDR(r.symbol) = ADDR(swap) = 0x80483c8

refaddr = ADDR(s) + r.offset = 0x80483bb

*refptr = (unsigned) (ADDR(r.symbol) +raddend – refaddr)

= (unsigned) (0x80483c8 + (-4) -0x80483bb)

= (unsigned) 0x9

(下一条 PC = refaddr + abs (raddend))

转移的偏移量 = ADDR(swap) - %rip

%rip = refaddr + abs(raddend)

80483ba: e8 09 00 00 00 call 80483c8 <swap>

Relocate absolute reference

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

      0: R_386_32 buf
```

ADDR(r.symbol) = ADDR(buf) = 0x8049454

*refptr = (unsigned) (ADDR(r.symbol) + raddend)
        = (unsigned) (0x8049454 + 0)
        =(unsigned) (0x8049454)

```
0804945c <bufp0>:
 804945c:      54 94 04 08
```

链接合并后的数据地址：

```
08049454 <buf>:
0804945c <bufp0>:
08049548 <bufp1>;
```

swap.o Relocation Info (.text)

```
extern int buf[];

int *bufp0 = &buf[0];

int *bufp1;

void swap()
{
  int temp;

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

```
08049454 <buf>:
0804945c <bufp0>:
08049548 <bufp1>;
```

```
Disassembly of section .text:

00000000 <swap>:
 0: 55                       push   %ebp
 1: 8b 15 5c 94 04 08        mov    0x0,%edx
                             3: R_386_32 bufp0
 7: a1 58 94 04 08            mov    0x4,%eax
                             8: R_386_32 buf
 c: 89 e5                     mov    %esp,%ebp
 e: c7 05 48 95 04 08 58      movl   $0x4,0x0
15: 94 04 08
                             10: R_386_32 bufp1
                             14: R_386_32 buf
18: 89 ec                    mov    %ebp,%esp
1a: 8b 0a                    mov    (%edx),%ecx
1c: 89 02                    mov    %eax,(%edx)
1e: a1 48 95 04 08           mov    0x0,%eax
                             1f: R_386_32 bufp1
23: 89 08                    mov    %ecx,(%eax)
25: 5d                       pop    %ebp
26: c3                       ret
```

Executable After Relocat

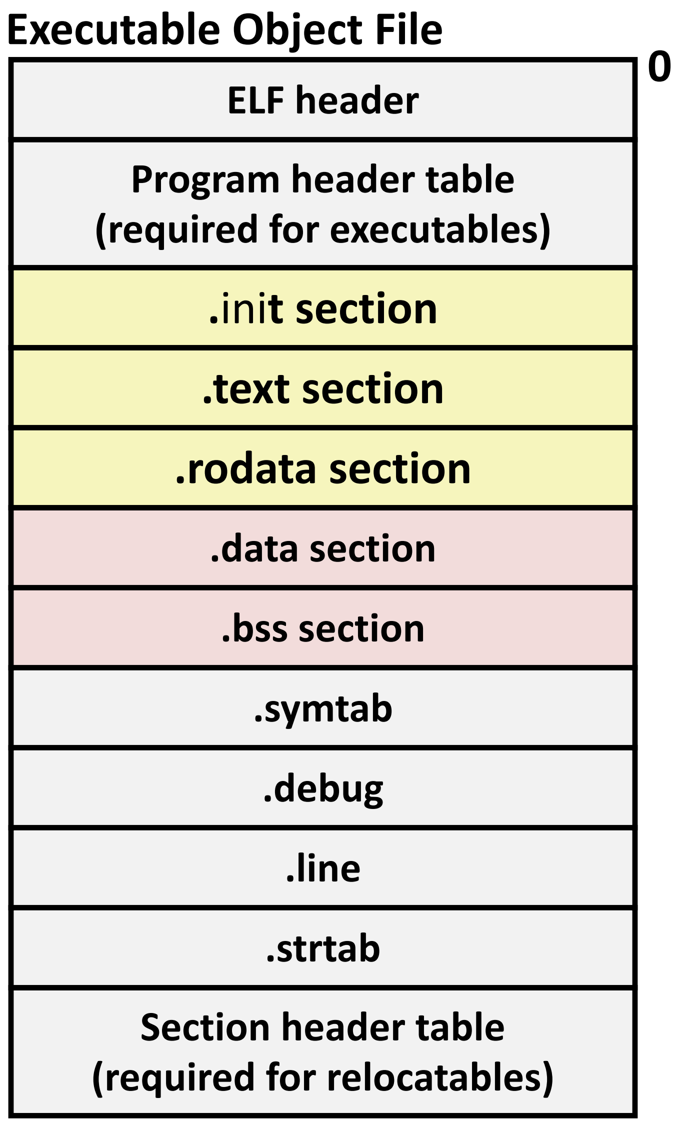
```
08049454 <buf>:
0804945c <bufp0>:
08049548 <bufp1>;
```

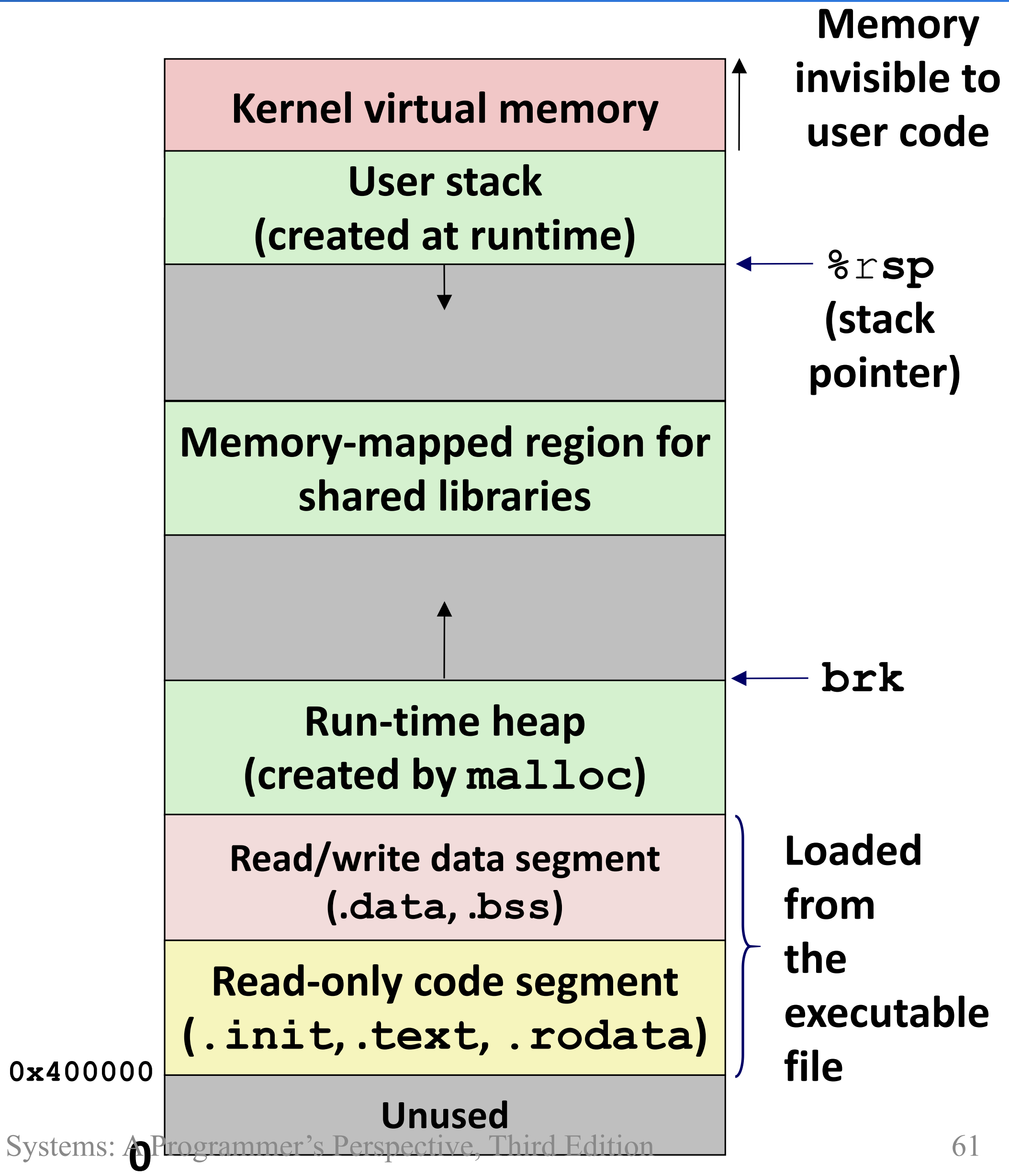
```
080483b4 <main>:
 80483b4:        55                              push   %ebp
 80483b5:        89 e5                           mov    %esp,%ebp
 80483b7:        83 ec 08                        sub    $0x8,%esp
 80483ba:        e8 09 00 00 00                  call   80483c8 <swap>
 80483bf:        31 c0                           xor    %eax,%eax
 80483c1:        89 ec                           mov    %ebp,%esp
 80483c3:        5d                              pop    %ebp
 80483c4:        c3                              ret
080483c8 <swap>:
 80483c8:        55                              push   %ebp
 80483c9:        8b 15 5c 94 04 08               mov    0x804945c,%edx
 80483cf:        a1 58 94 04 08                  mov    0x8049458,%eax
 80483d4:        89 e5                           mov    %esp,%ebp
 80483d6:        c7 05 48 95 04 08 58            movl   $0x8049458,0x8049548
 80483dd:        94 04 08
 80483e0:        89 ec                           mov    %ebp,%esp
 80483e2:        8b 0a                           mov    (%edx),%ecx
 80483e4:        89 02                           mov    %eax,(%edx)
 80483e6:        a1 48 95 04 08                  mov    0x8049548,%eax
 80483eb:        89 08                           mov    %ecx,(%eax)
 80483ed:        5d                              pop    %ebp
 80483ee:        c3                              ret
```

Executable After Relocation (.data)

```
Disassembly of section .data:

08049454 <buf>:
 08049454:       01 00 00 00 02 00 00 00

0804945c <bufp0>:
 0804945c:       54 94 04 08
```

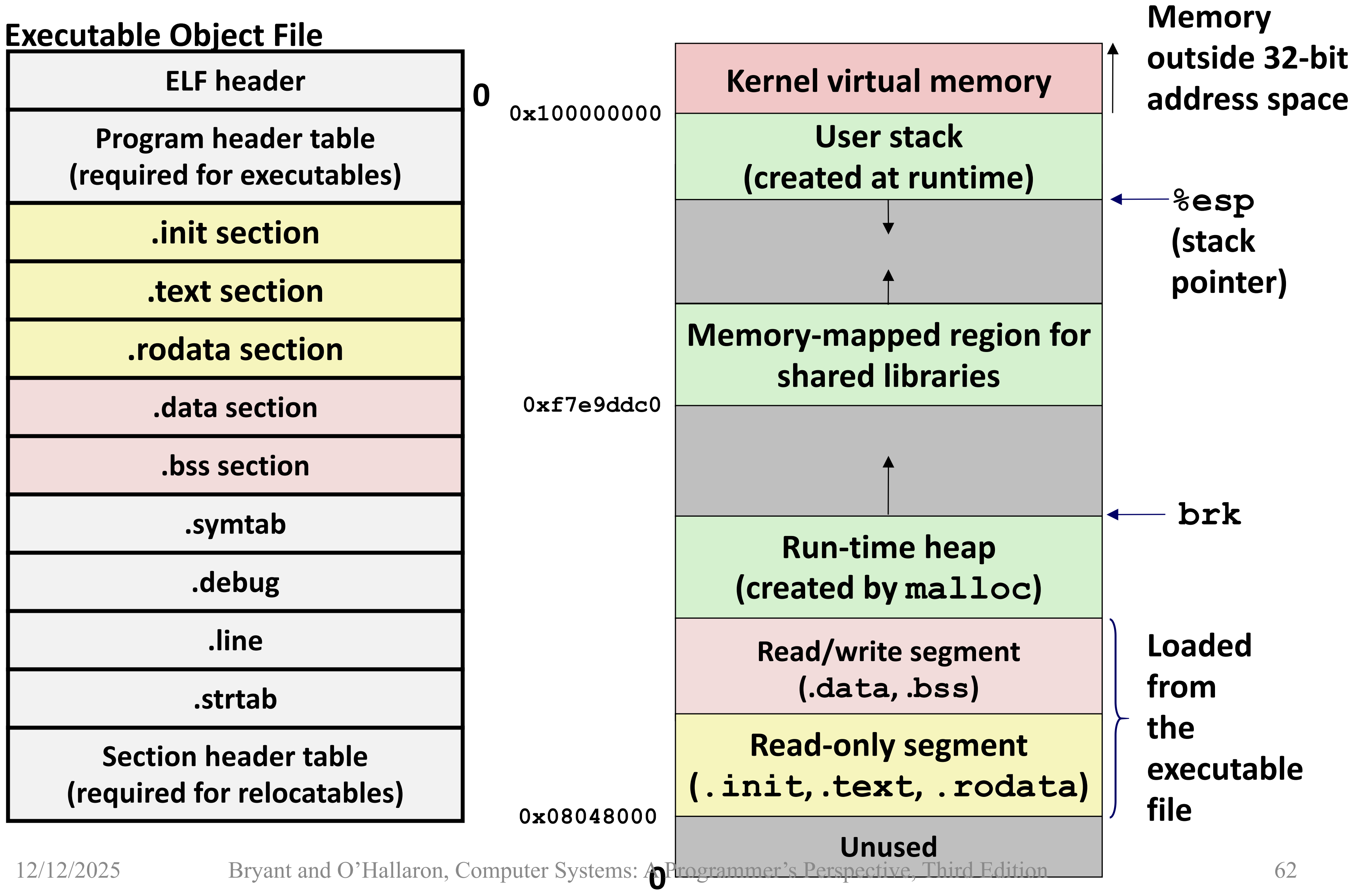
Loading Executable Object Files(64)





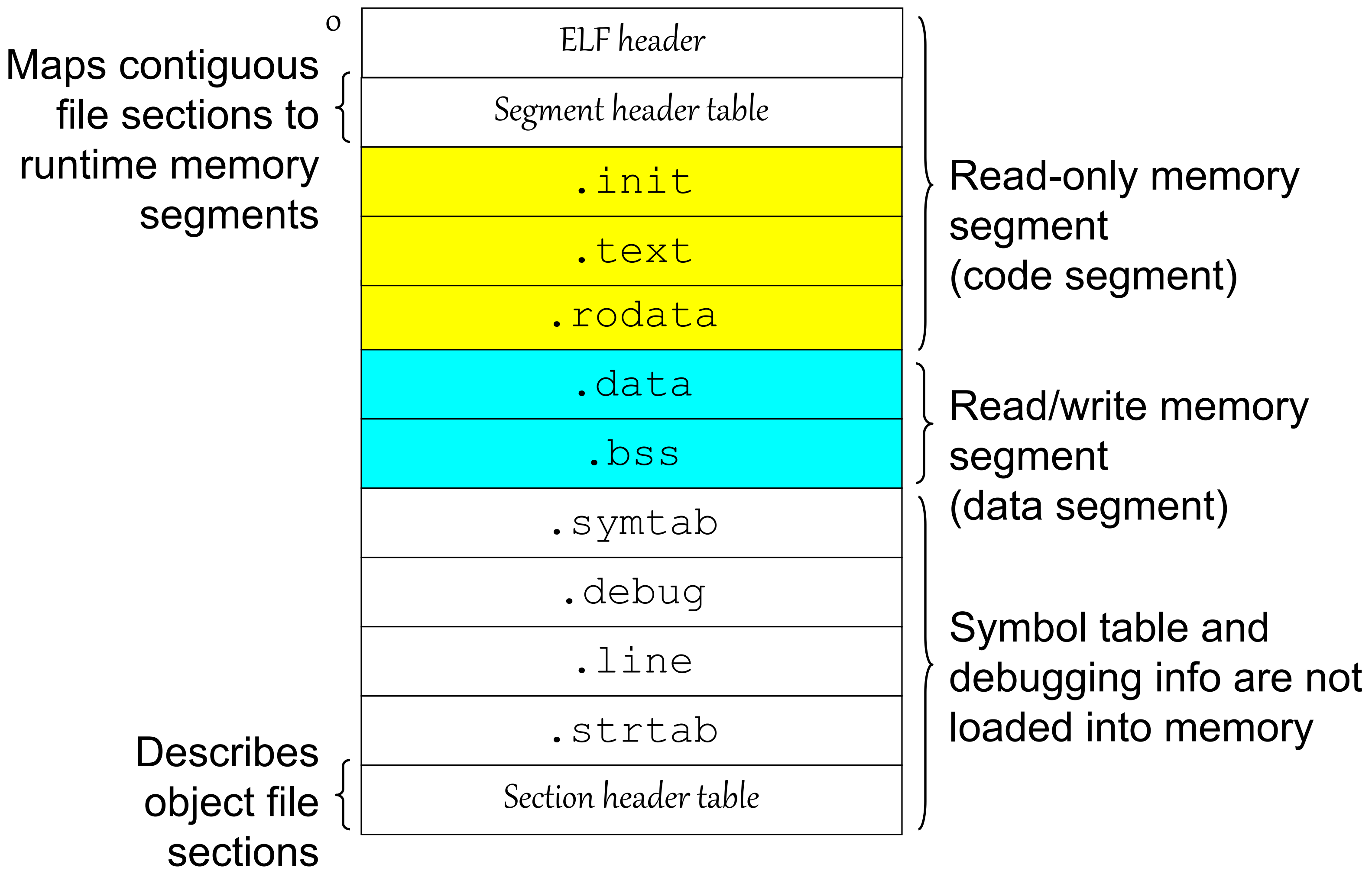
Kernel virtual memory
Memory invisible to user code
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write data segment (.data, .bss)
Read-only code segment (.init, .text, .rodata)
Loaded from the executable file
0x400000
Unused
0

Loading Executable Object Files(32）



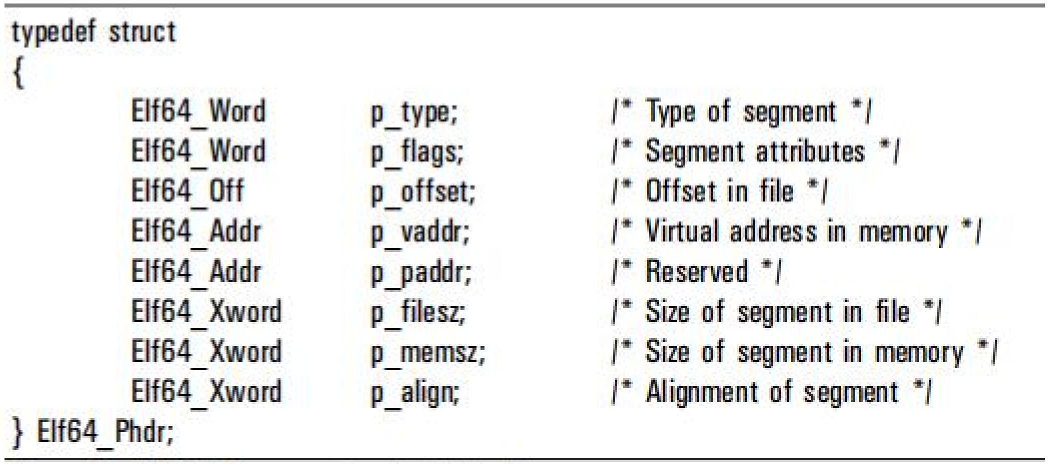
Executable Object File
ELF header
Program header table (required for executables)
.init section
.text section
.rodata section
.data section
.bss section
.symtab
.debug
.line
.strtab
Section header table (required for relocatables)
0
0x100000000
Kernel virtual memory
Memory outside 32-bit address space
User stack (created at runtime)
%esp (stack pointer)
Memory-mapped region for shared libraries
0xf7e9ddc0
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
0x08048000
Unused
0

Executable Object File Format


0
ELF header
Segment header table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section header table
Maps contiguous file sections to runtime memory segments
Describes object file sections
Read-only memory segment (code segment)
Read/write memory segment (data segment)
Symbol table and debugging info are not loaded into memory

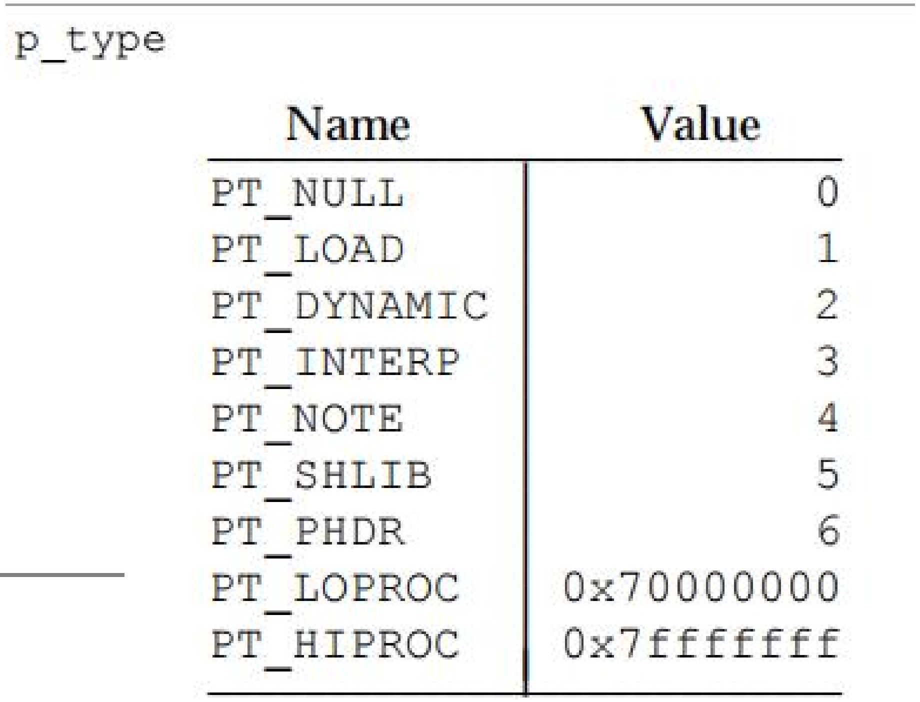
Program Header

```
  typedef  struct {
        Elf 32_Word     p_type ;
        Elf32_Off       p_offset ;
        Elf32_Addr   p_vaddr;
        Elf32_ Addr  p_paddr;
        Elf32_ Word  p_filesz ;
        Elf32_ Word  p_memsz ;
        Elf32_ Word  p_flags ;
        Elf32_ Word  p_align ;
} Elf32_Phdr;
```



```
typedef struct
{
        Elf64_Word      p_type;     /* Type of segment */
        Elf64_Word      p_flags;    /* Segment attributes */
        Elf64_Off       p_offset;   /* Offset in file */
        Elf64_Addr      p_vaddr;    /* Virtual address in memory */
        Elf64_Addr      p_paddr;    /* Reserved */
        Elf64_Xword     p_filesz;   /* Size of segment in file */
        Elf64_Xword     p_memsz;    /* Size of segment in memory */
        Elf64_Xword     p_align;    /* Alignment of segment */
} Elf64_Phdr;
```

Figure 6. ELF-64 Program Header Table Entry



p_type

Name	Value
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

Segment head table

- Read-only code segment

```
LOAD off 0x00000000     vaddr 0x08048000    paddr 0x08048000 align 2**12
         filesz 0x00000448   memsz 0x00000448   flags r-x
```

- Read/write data segment

```
LOAD off 0x00000448     vaddr 0x08049448    paddr 0x08049448 align 2**12
         filesz 0x000000e8   memsz 0x00000104   flags rw-
```

```
readelf -h  /bin/ps                                  // 显示目标文件的ELF header
readelf -l  /bin/ps                                  // 显示目标文件的Program header table
readelf -S /bin/ps   or  objdump -h  /bin/ps          // 显示目标文件的Section header table
objdump -f /bin/ps                                   //显示目标文件的file header
objdump -x /bin/ps                                   //显示所有的headers
```

linking

- **链接技术核心基础**
  - 链接的动机与核心价值
  - 目标文件的分类
  - 链接器的核心功能
- **符号解析与重定位**
- **代码重用**
  - 静态链接库
  - 动态链接库
- **链接器的应用（价值体现）**
  - 程序的模块化构建
  - 符号控制与冲突消解
  - 代码复用
  - 调试与监控：库插桩技术

Libraries:Packaging Commonly Used Functions

- **如何打包程序员常用的函数？（功能复用问题）**
  - Math, I/O, memory management, string manipulation, etc.

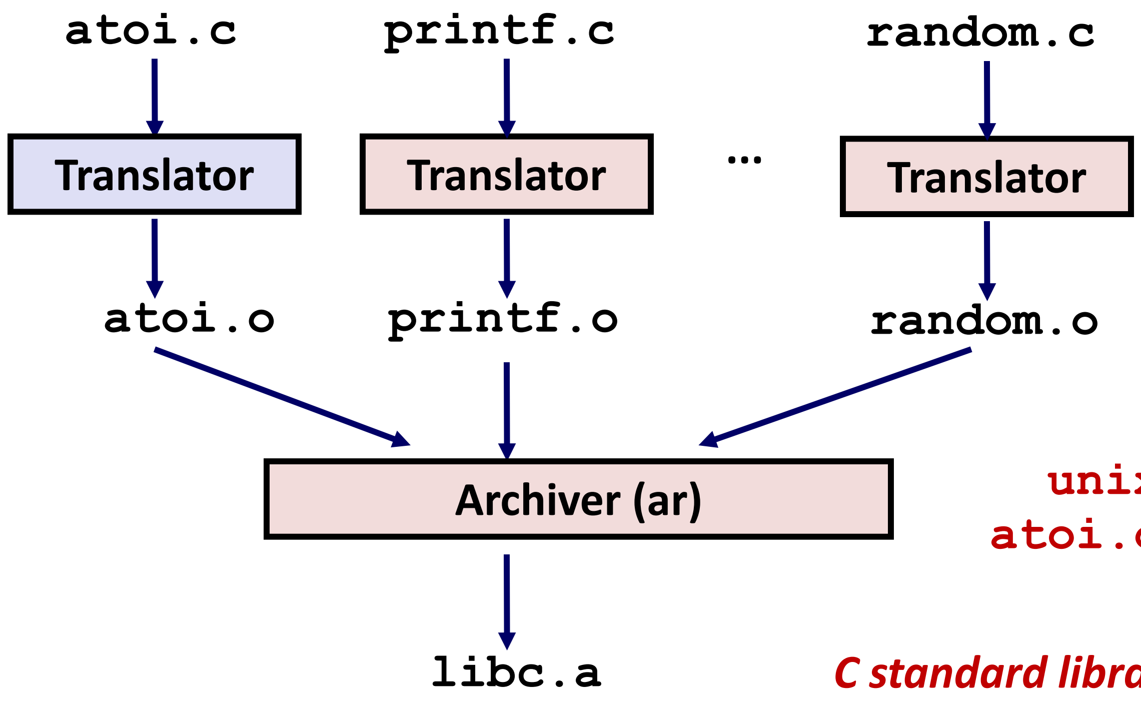
- **在没有库概念的情况下，鉴于目前的链接器框架：**
  - Option 1: 将所有常用函数放入一个源文件中
    - 程序员将该文件链接到他们的程序中
    - 空间和时间效率低下
  - Option 2: 将每个函数放在单独的源文件中
    - 程序员明确地将适当的二进制文件链接到他们的程序中
    - 更高效，但对程序员来说负担较重

Old-fashioned Solution: Static Libraries

- **静态链接库(Static libraries) (`.a` archive files)**
  - 将相关的可重定位对象文件合并成一个带有索引的单一文件 (called an *archive*)
  - 增强连接器功能，使其能够尝试通过在一个或多个存档（库文件）中查找符号来解决未解析的外部引用
  - 如果存档成员文件解析了引用，将其所在的模块链接到可执行文件中

Creating Static Libraries

如何创建库文件?



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

C standard library

- 库文件可以增量更新
- 可以将修改后的函数重新编译后替换掉库文件中对应函数

Static Libraries Format

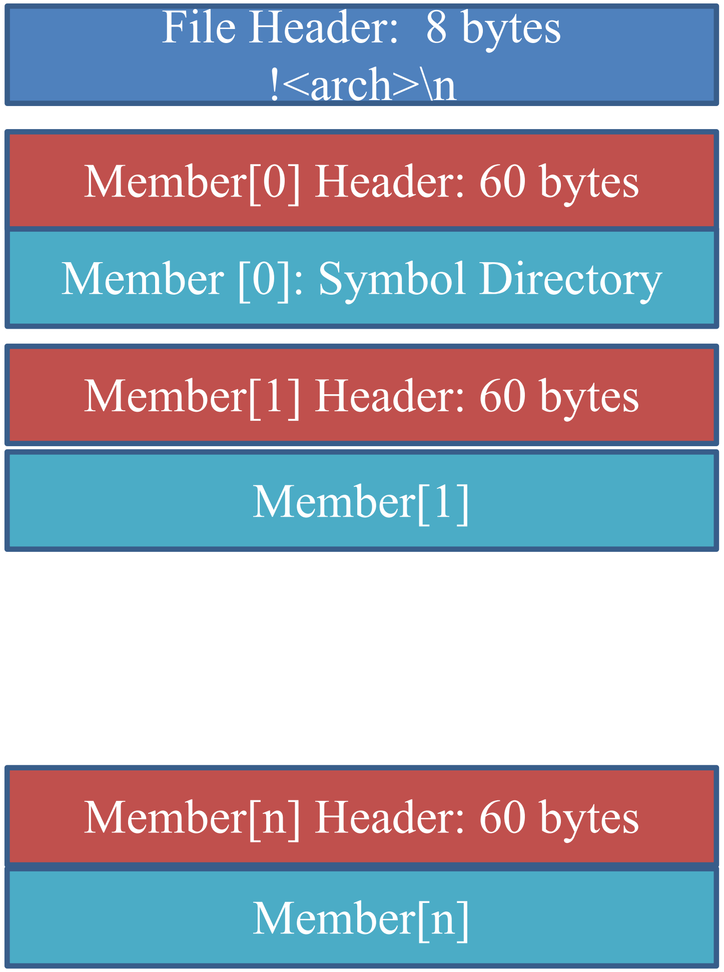
- **Unix archive format**
  - File header: !<arch>\n
  - Member header:

```
char name[16];  /*member name */
char modtime[12]; /*modification time*/
char uid[6];  /* user ID */
char gid[6];  /* group ID */
char mode[8]; /*octal  file mode */
char size[10];  /*member size */
char eol[2];  /*reverse quote, newline*/
```

  - Symbol Directory (ELF)

```
int nsymbols;  /*number of symbols*/
int member[];  /*member offsets */
char string[];
```

库目录必须是库中的第一个成员，其名称（name）为 / （ASCII 2f)


File Header: 8 bytes
!<arch>\n
Member[0] Header: 60 bytes
Member [0]: Symbol Directory
Member[1] Header: 60 bytes
Member[1]
Member[n] Header: 60 bytes
Member[n]

Linking with Static Libraries

libvector.a

```
#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

```
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

vector.h 声明在其他模块中定义的函数

libvector.a

- 创建库libvector.a
  – ar rs libvector.a addvec.o multvec.o
- 查看二进制libvector.a 库文件前68个字节
  – hexdump -n 160 libvector.a

file header !<arch>\n

```
0000000 3c21 7261 6863 0a3e  |202f 2020 2020 2020
0000010 2020 2020 2020 2020 |2030 2020 2020 2020
0000020 2020 2020 |2030 2020 2020 |2030 2020 2020
0000030 2030 2020 2020 2020 |3035 2020 2020 2020
0000040 2020| 0a60 0000 0200 0000 6000 0000 5c06
0000050 6461 7664 6365 6d00 6c75 7675 6365 0000

0000060 6461 7664 6365 6f2e  202f  2020 2020 2020
0000070 2030 2020 2020 2020 2020 2020 2030 2020
0000080 2020 2030 2020 2020 3436 2034 2020 2020
0000090 3431 3237 2020 2020 2020 0a60  457f 464c
00000a0
```

member0 header
库目录成员头，
name：2f (/)

member0：
库目录

member1 header
name: addvec.o2f(/)

member 1

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)

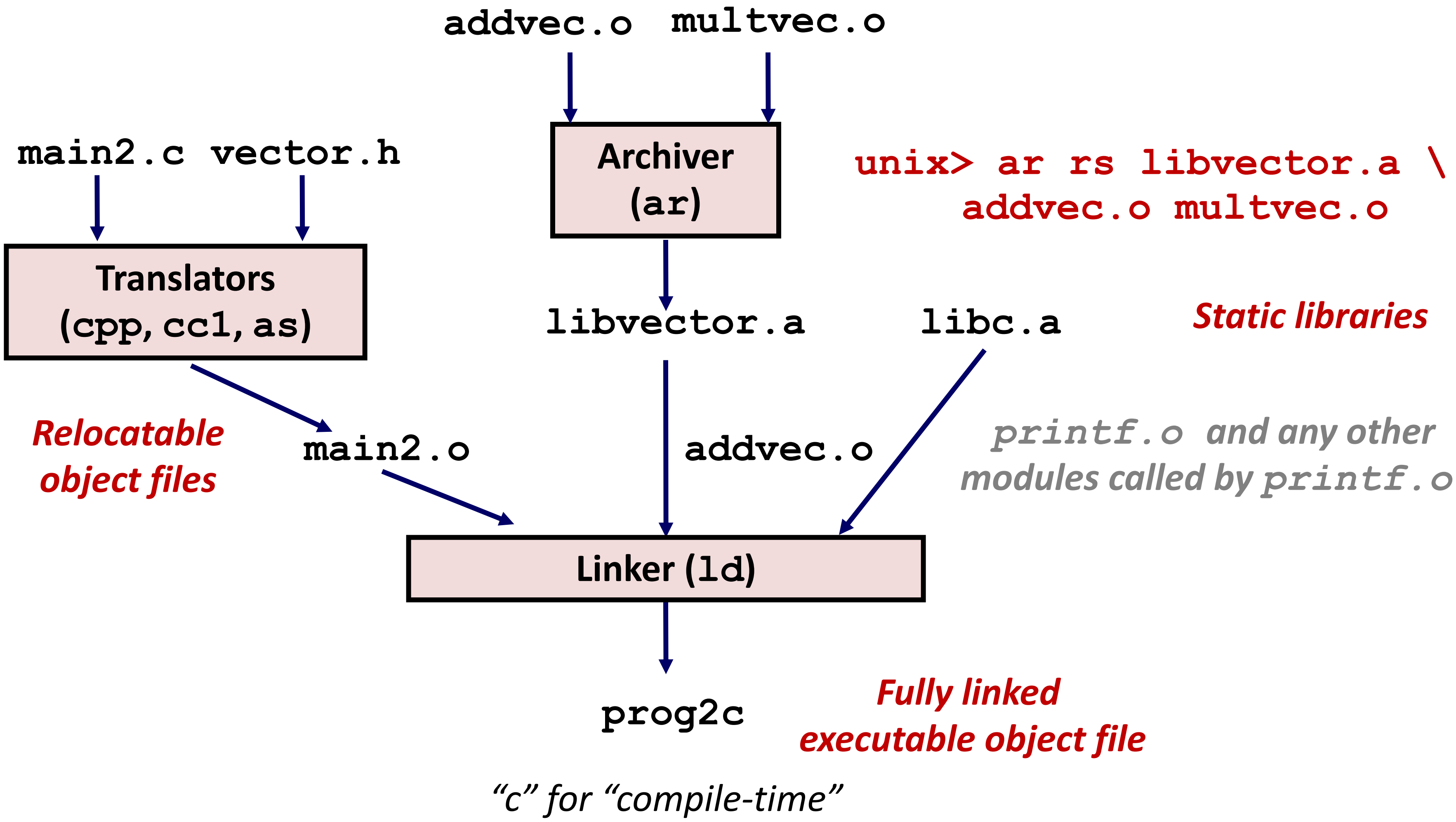
- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t 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
…
```

Linking with Static Libraries

如何链接库文件



```
unix> ar rs libvector.a \
        addvec.o multvec.o
```

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

Using Static Libraries

- **用于解析外部引用的链接器算法:**
  - Scan `.o` files and `.a` files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.
- **问题:**
  - **Command line order matters!**
  - Moral: put libraries at the end of the command line.

```
unix> gcc -static -o prog2c -L. -lvector main2.o
main2.o: In function `main':
main2.c:(.text+0x19): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

Using Static Libraries

U：未被解析的符号引用集合
D：已经定义的符号
E: 将被链接的目标文件（模块）

链接器用于解析外部引用的算法:

- **从左到右扫描命令行中的 .o files 和 .a files**
- **在扫描过程中将未解析的符号引用加入U集合，将要被链接的目标文件加入E集合，将已经定义的符号加入D集合**
- **Initially, E,U,D are empty.**
- **对于每一个命令行中的输入文件 f**
  - 如果**f是.o目标文件**, 将f加入E集合，并根据符号表和f中的引用，更新U集合和D集合, proceed to next
  - 如果**f是.a 库文件,** 链接器将U中的符号与库文件中**成员模块m**定义的符号进行匹配操作
    - 如果**成员模块m**定义的符号与U中的符号匹配，将模块m加入E，根据m的符号定义和引用更新U 和 D
    - 迭代执行上一步，直到U和D稳定不再有更新，并丢弃该库文件中不在E集合的其他成员模块 proceed to next
- **如果命令行扫描结束后U不为空，则报错；否则链接器将E中的目标文件合并 并重定位符号引用，输出产生可执行文件**

Using Static Libraries

- **Problem:**
  - **Command line order matters!**
  - Moral: put libraries at the end of the command line.

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

- **If libraries are not independent （存在库依赖）**
  - **要求**：对于每个库成员引用的符号s, 在命令行中至少要跟随一个该符号的定义
  - 假设 foo.c 调用libx.a和libz.a中的函数，并且libz.a 中的被调用函数又调用liby.a中的函数
    - Unix> gcc foo.c libx.a libz.a liby.a
  - 在命令行中，为了满足库依赖的需求，可以重复链接库文件。例如 foo.c 调用libx.a中的函数，该函数又调用liby.a中的函数，这个被调用的函数又需要调用libx.a中的函数，则
    - Unix> gcc foo.c libx.a liby.a libx.a

Linking Review

- **链接是一种允许程序由多个目标文件（Object file)构建的技术**
  - Motivation：模块化设计、构建应用程序的高效性
  - What it does：符号解析、重定位
- **How it works: 两遍扫描**
  - Pass1：读取section大小，计算最终的(虚拟)内存布局。同时，读入所有符号，在(虚拟)内存中创建完整的符号表。
  - Pass2：读section和重定位信息、更新地址、写新文件(Executable/loadable)
- **链接过程可以在程序生命周期的不同阶段进行：**
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)
- **理解链接技术可以帮助你避免讨厌的错误，并使你成为一名更好的程序员**

Outline

- **链接技术核心基础**
  - 链接的动机与核心价值
  - 目标文件的分类
  - 链接器的核心功能
- **符号解析与重定位**
- **代码重用**
  - 静态链接库
  - **动态链接库**
- **链接器的应用**
  - 程序的模块化构建
  - 符号控制与冲突消解
  - 代码复用
  - 调试与监控：库插桩技术

Linking

- **Linking**
  - **Motivation：模块化设计、构建应用程序的高效性**
  - **What it does：符号解析、重定位**
  - **How it works: 两遍扫描**
    - **Pass1：读取section大小，计算最终的内存布局。同时，读入所有符号，在内存中创建完整的符号表。**
    - **Pass2：读取section和重定位信息，更新地址，写新文件（Executable/loadable）。**
  - **Dynamic linking**
- **Position-Independent Code (PIC)**
- **Case study: Library interpositioning**

Modern Solution: Shared Libraries

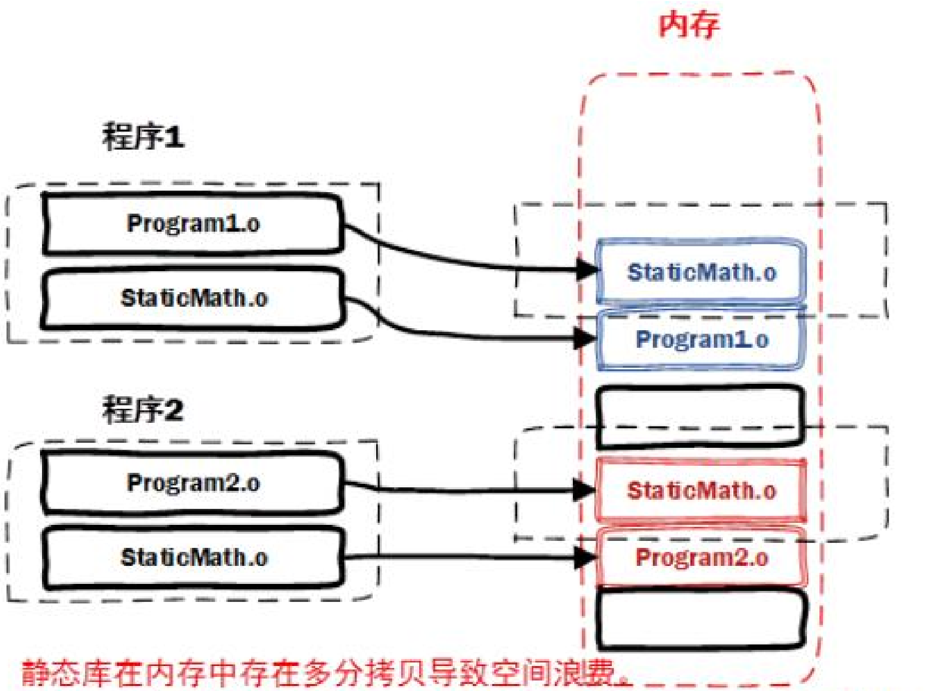
- **静态链接库的缺点:**
  - Duplication in the stored executables (every function needs libc) **（文件）**
  - Duplication in the running executables **（内存映像）**
  - Minor bug fixes of system libraries require each application to explicitly relink **（维护）**
    - Rebuild everything with glibc?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

- **目前解决方案: 可共享库(Shared Libraries)**
  - Object files that contain code and data that are loaded and linked into an application ***dynamically***, at either ***load-time*** or ***run-time***
  - Also called: dynamic link libraries, DLLs, `.so` files

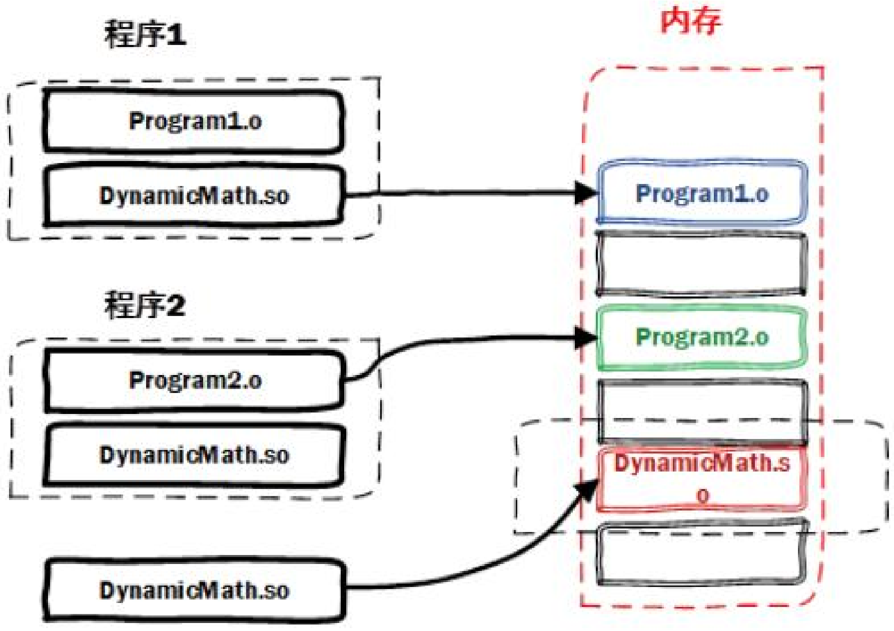
Shared Libraries (cont.)

- **动态链接可以在可执行文件第一次加载并运行时发生(加载时链接)**
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.
- **动态链接也可以在程序开始运行后发生(运行时链接)**
  - In Linux, this is done by calls to the **`dlopen()`** interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.
- **共享库例程可以由多个进程共享**
  - More on this when we learn about virtual memory

静态链接库vs. 动态链接库



静态库在内存中存在多分拷贝导致空间浪费。
假如，静态库占用1M内存，有2000个这样的程序，将占用近2GB的空间~~~~~



动态库在内存中只存在一份拷贝，避免了静态库浪费空间的问题。

What dynamic libraries are required?

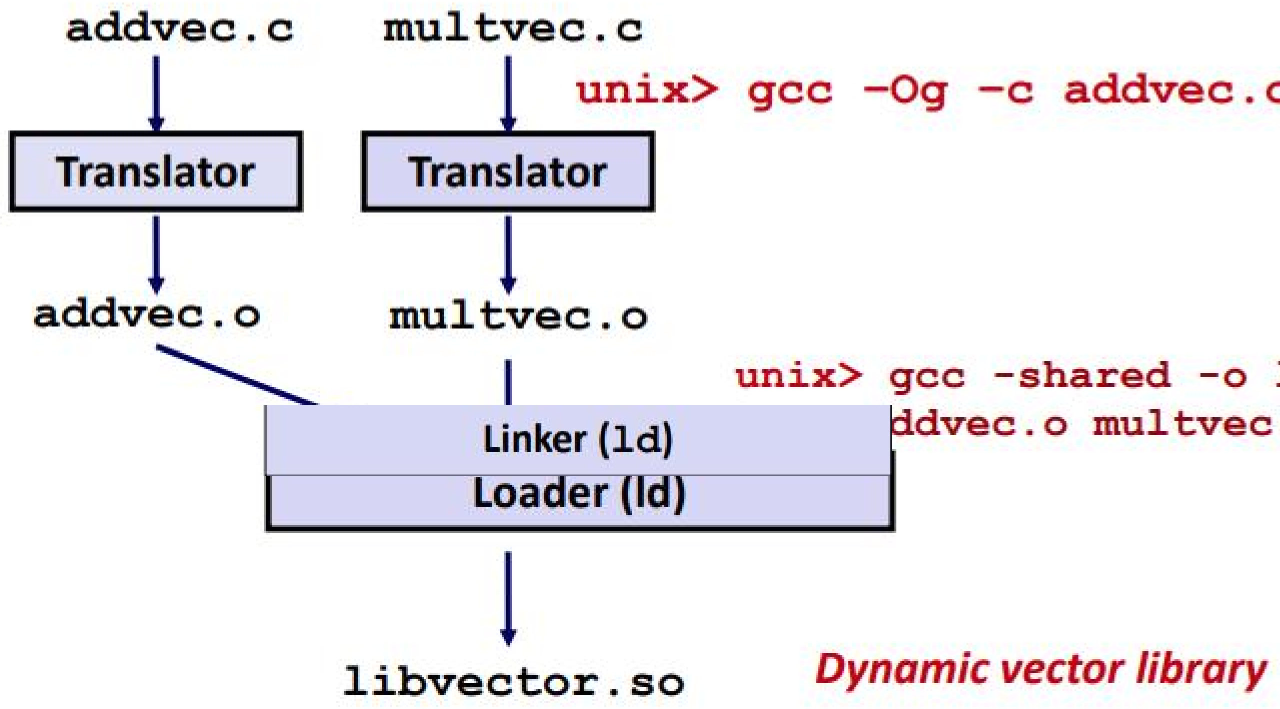
- **.interp section**
  - 指定使用的动态链接器 (i.e., ld-linux.so)
- **.dynamic section**
  - 指定要使用的动态库的名称等信息
  - Follow an example of prog

```
(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

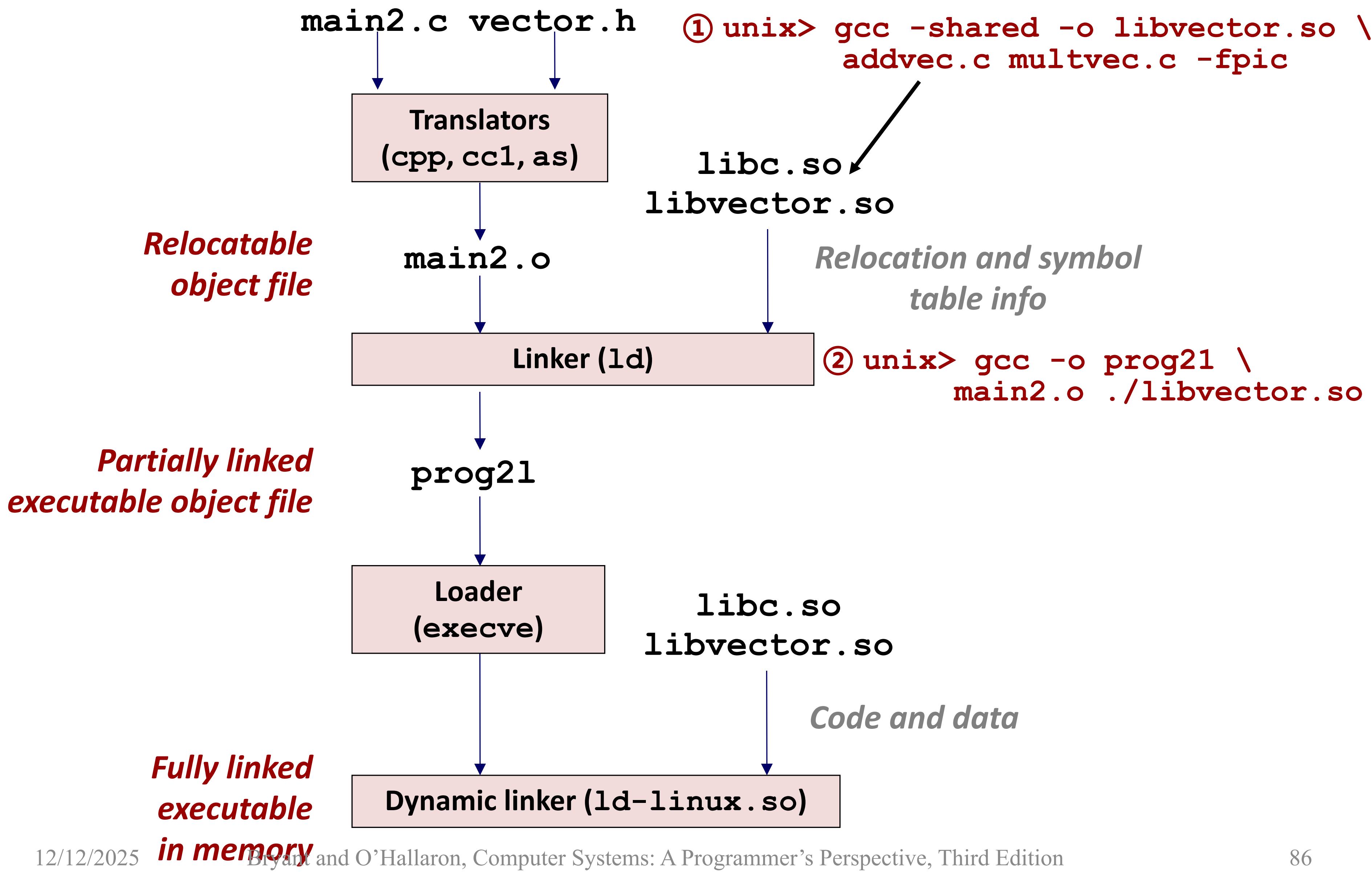


```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

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

生成动态链接库libvector.so

Dynamic Linking at Load-time


main2.c vector.h
Translators (cpp, cc1, as)
Relocatable object file
main2.o
libc.so
libvector.so
Relocation and symbol table info
Linker (ld)
Partially linked executable object file
prog21
Loader (execve)
libc.so
libvector.so
Code and data
Fully linked executable in memory
Dynamic linker (ld-linux.so)

```
① unix> gcc -shared -o libvector.so \
        addvec.c multvec.c -fpic
```

```
② unix> gcc -o prog21 \
        main2.o ./libvector.so
```

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()
{
    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);
    }
#next page
```

void *dlopen(const char *filename, int flag);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
char *dlerror(void);

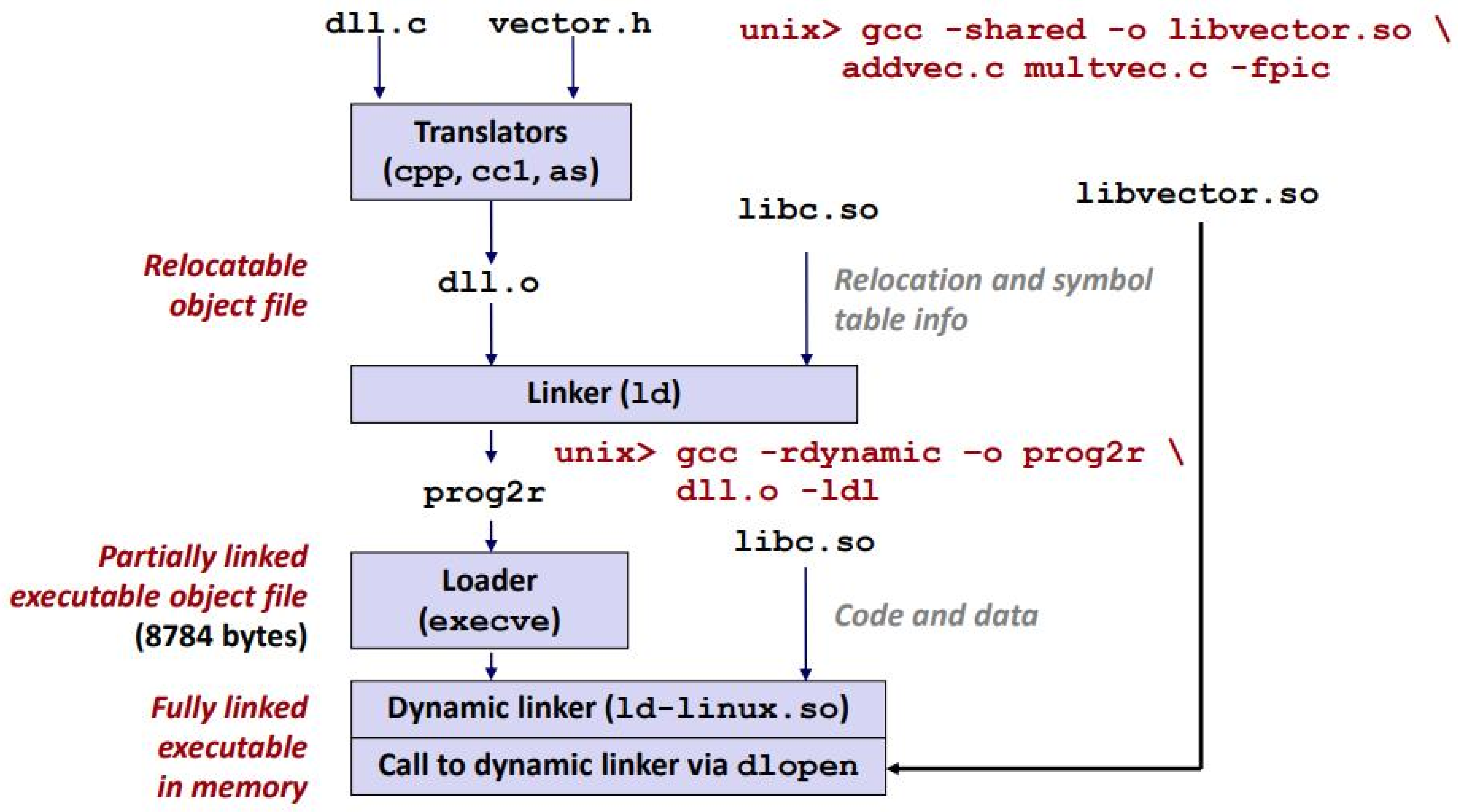
dll.c

Dynamic Linking at Run-time

```
    ...

    /* 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
libvector.so
Relocation and symbol table info
Linker (ld)
prog2r
Partially linked executable object file (8784 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
```

Linking

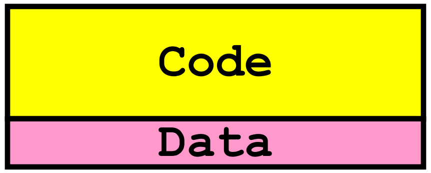
- Linking
  - Motivation
  - What it does
  - How it works
  - Dynamic linking
- **Position-Independent Code (PIC)**
- Case study: Library interpositioning

Position-Independent Code(PIC)

- **可共享库的主要目的：**
  - 允许正运行的多个进程共享内存中相同的库代码，以节约内存资源
- **多个进程如何共享一个程序的单一副本?**
  - One approach: 分配一块专用的地址空间
    - Inefficient use of address space
    - Difficult to manage
- **用 PIC code: (库)代码可以在任何地址加载和执行，而不需要被链接器再重定位(relocation).**
  - Calls to procedure in the same object module are already PIC, because the references are PC-relative
  - **Calls to externally defined procedures and references to global variables are not normally PIC.**
- **PIC代码：可以加载而无需重定位的代码**
  - gcc 中 使用 **- fpic 选项**

PIC-Global data (1/2)

观察到的现象: 数据段总是紧跟在代码段之后加载，因此代码段中的任何一条指令与数据段中的任何变量之间的相对距离是常数，在运行时保持不变。
编译器在数据段的起始位置创建一个GOT(Global offset table)，GOT包含object module引用的每个全局数据对象的entry，并且编译器为GOT中的每个entry生成重定位记录，在加载过程时，动态链接器重定位每个GOT entry，使得每个GOT的entry包含要引用的对象的绝对地址。



Relative offset is constant
Add GOT at begin of Data segment

- **Global data reference:**

```
        call L1
L1:   popl %ebx                  ebx contains the current PC
      addl $VAROFF,%ebx          ebx points to the GOT entry for var
      movl (%ebx), %eax          reference indirect through the GOT
      movl (%eax), %eax
```

PIC-Global data (2/2)

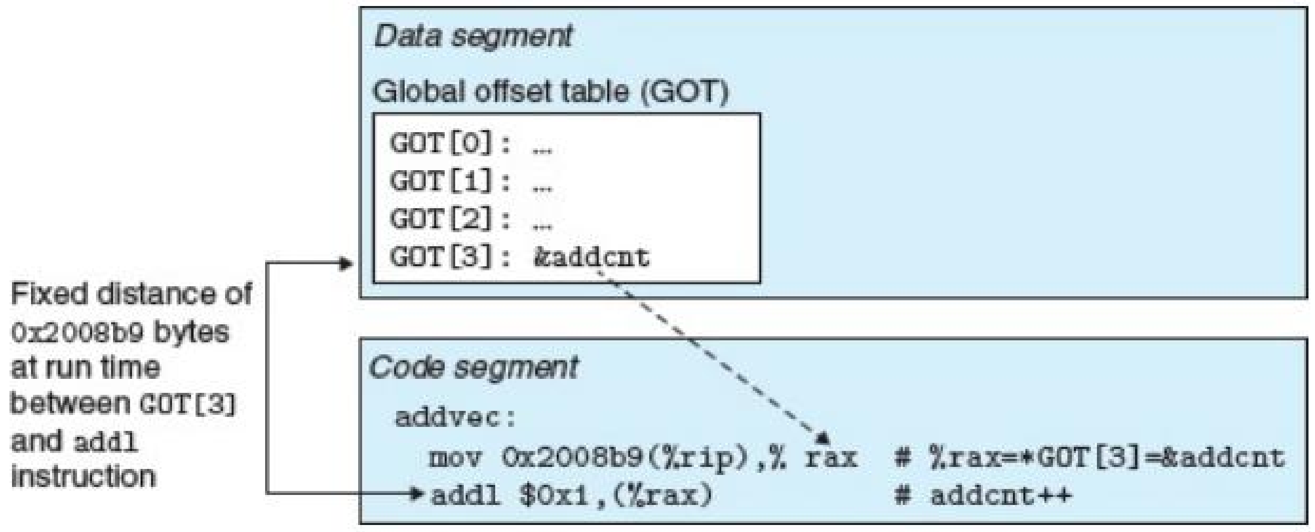

Data segment
Global offset table (GOT)
GOT[0]: …
GOT[1]: …
GOT[2]: …
GOT[3]: &addcnt
Fixed distance of 0x2008b9 bytes at run time between GOT[3] and addl instruction
Code segment
addvec:
mov 0x2008b9(%rip),% rax # %rax=*GOT[3]=&addcnt
addl $0x1,(%rax) # addcnt++

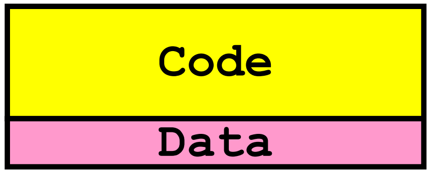
Figure 7.18 Using the GOT to reference a global variable. The `addvec` routine in `libvector.so` references `addcnt` indirectly through the GOT for `libvector.so`.

```
X86_64:
movq VAROFF(%rip),%rax
movl (%rax),%eax
```

PIC-Global Procedure

GOT (Global offset table, GOT)

```
        call L1
L1:     popl %ebx
        addl $PROCOFF,%ebx
        call  *(%ebx)
```



Relative offset is constant

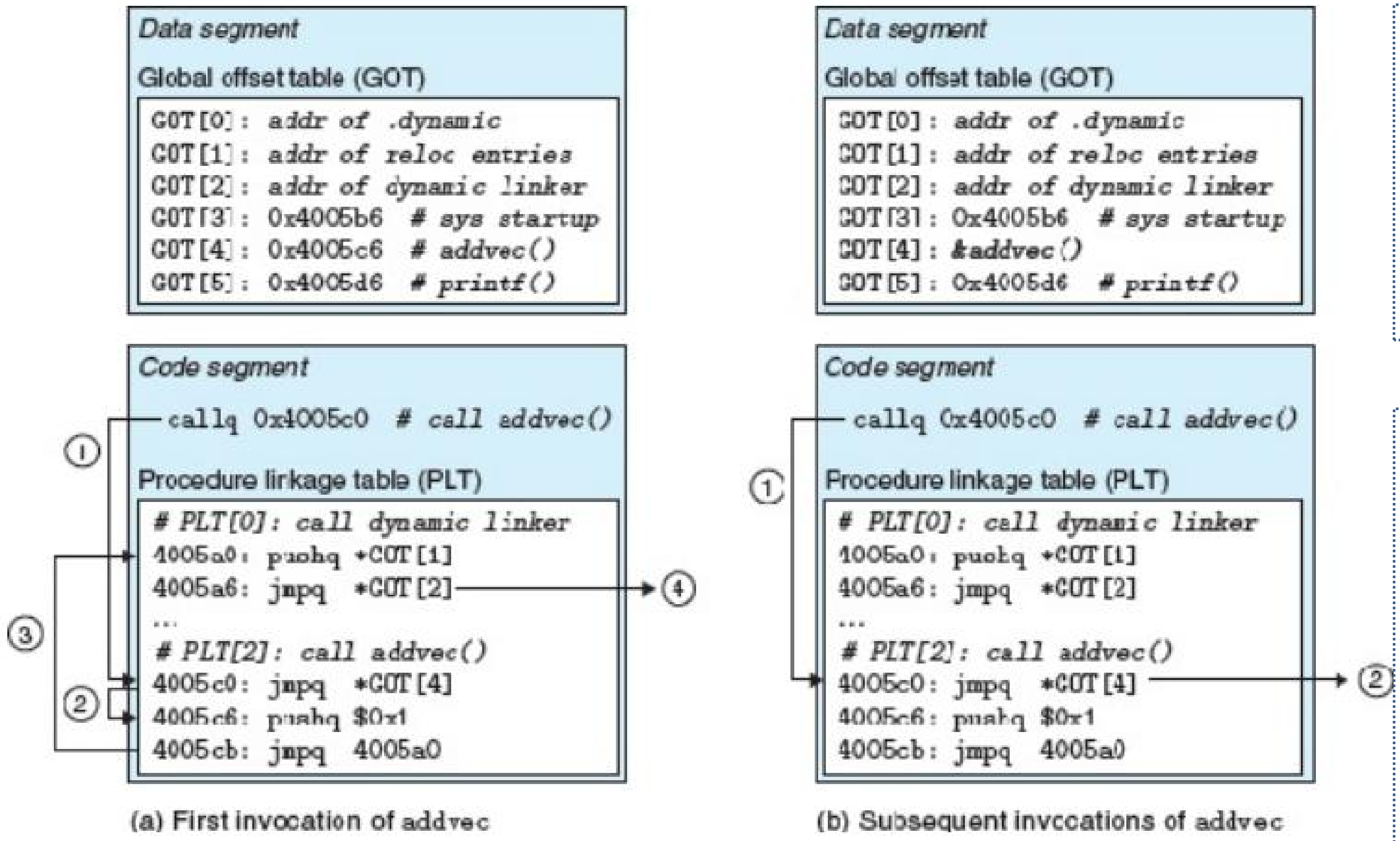
Add GOT at begin of Data segment

一种方式：与全局数据一样在库加载时解析和绑定该库中所有函数的地址，可能很多函数并不会被调用）

另一种方式：ELF编译系统使用Lazy binding技术即延迟绑定函数地址，直到首次调用该函数时才进行。

实现方式： GOT + PLT（Procedure linkage table)
如果一个对象模块调用了任何在共享库中定义的函数，则它将拥有自己的GOT和PLT 。GOT是.data section的一部分。 PLT 是.text section的一部分。

PIC-Global Procedure Call（1/2)



Data segment
Global offset table (GOT)
Code segment
Procedure linkage table (PLT)
(a) First invocation of addvec
(b) Subsequent invocations of addvec

(a)

```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: 0x4005c6  # addvec()
GOT[5]: 0x4005d6  # printf()

callq 0x4005c0  # call addvec()

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

(b)

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

callq 0x4005c0  # call addvec()

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

GOT[0]和GOT{1]：
动态链接器所需信息
GOT[2]:
动态链接器的地址
GOT[4]:
所需调用的函数地址

PLT[0]:
调用动态链接器
PLT[1]:
调用Startup 函数，初始化运行环境，调用main函数，处理返回值
PLT[2]:
调用addvec()

Figure 7.19 Using the PLT and GOT to call external functions. The dynamic linker resolves the address of `addvec` the first time it is called.

PIC-Global Procedure Call（2/2)

- **addvec()在第一次被调用时，GOT和PLT如何一起工作来解析addvec函数的运行时地址?**
  - step1: 控制流首先转到PLT[2]，PLT[2]的第一条指令是跳转指令，该指令跳转地址与addvec()对应的GOT条目（GOT[4]）
  - step2: 根据GOT[4]的内容实现间接跳转，跳转回PLT[2]的下条指令
  - step3: 将addvec()的标识压栈，然后跳转到PLT[0]
  - step4: PLT[0] 首先基于GOT[1]中的重定位参数压栈，然后通过GOT[2]调用动态链接器，链接器根据step3和step4的压栈参数重定位addvec()的运行时地址，并挑转到addvec()
- **addvec()在后续被调用时的控制流**
  - step1: 控制流首先跳转到PLT[2]
  - step2: 通过GOT[4]间接跳转到addvec()

Outline

- **链接技术核心基础**
  - 链接的动机与核心价值
  - 目标文件的分类
  - 链接器的核心功能
- **符号解析与重定位**
- **代码重用**
  - 静态链接库
  - 动态链接库
- **链接器的应用（价值体现）**
  - 程序的模块化构建
  - 符号控制与冲突消解
  - 代码复用
  - **调试与监控：库插桩技术**

Case Study: Library Interpositioning

- **库插桩技术(Library interpositioning)：强大的链接技术**
  - 允许程序员拦截对函数的调用，取而代之执行自己的代码
  - 可追踪对某个函数的调用次数，验证和追踪它的输入和输出
  - 甚至替换成一个完全不同的实现
- **插桩的时点：**
  - **Compile time:** 源代码被编译时
  - **Link time:** 当可重定位对象文件通过静态链接组合成可执行对象文件时
  - **Load/run time:** 当可执行对象文件被载入内存并动态链接后执行时

Some Interpositioning Applications

- **增强安全性 (Security)**
  - Confinement (sandboxing)
  - Behind the scenes encryption
- **增强调试能力 (Debugging)**
  - In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
  - Code in the SPDY networking stack was writing to the wrong location
  - Solved by intercepting calls to Posix write functions (write, writev, pwrite)

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

Some Interpositioning Applications

- 监控和性能分析 (Monitoring and Profiling)
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - Malloc tracing
    - Detecting memory leaks
    - Generating address traces
- 错误校验 (Error Checking)
  - C Programming Lab used customized versions of malloc/free to do careful error checking
  - Other labs (malloc, shell, proxy) also use interpositioning to enhance checking capabilities

Example program

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

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

- **目标:**
  - trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.

- **三种解决方案:**
  - interpose on the `lib` `malloc` and `free` functions at **compile time, link time, and load/run time.**

Compile-time Interpositioning

```
#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

Compile-time Interpositioning

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

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

Local 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>
```

Link-time Interpositioning

```
#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

Link-time Interpositioning

```
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" 选项将参数传递给链接器，将每个逗号替换为空格**
- **“--wrap,malloc”参数指示链接器以特殊方式解析引用:**
  - 引用 malloc 解析为 __wrap_malloc (两个下划线)
  - 引用__real_malloc应该解析为malloc

Load/Run-time Interpositioning

Load/Run-time Interpositioning

```
#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

Load/Run-time Interpositioning

```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

Load/Run-time Interpositioning

```
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>
```

- **The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `mymalloc.so` first.**
- **Type into (some) shells as:**

  ```
  env LD_PRELOAD=./mymalloc.so ./intr)
  ```

Interpositioning Recap

- **Compile Time**
  - Apparent calls to malloc/free get macro-expanded into calls to mymalloc/myfree
- **Link Time**
  - Use linker trick to have special name resolutions
    - malloc → __wrap_malloc
    - __real_malloc → malloc
- **Load/Run Time**
  - Implement custom version of malloc/free that use dynamic linking to load library malloc/free under different names
  - Can use with ANY dynamically linked binary

```
env LD_PRELOAD=./mymalloc.so gcc -c int.c
```

Linking Recap

- **Usually: Just happens, no big deal**
- **Sometimes: Strange errors**
  - Bad symbol resolution
  - Ordering dependence of linked .o, .a, and .so files
- **For power users:**
  - Interpositioning to trace programs with & without source

Acknowledgements

- **This course was developed and fine-tuned by Randal E. Bryant and David O'Hallaron. They wrote *The Book*!**
- **http://www.cs.cmu.edu/~./213/schedule.html**

PIC-Global Procedure Call

- Lazy binding: GOT + PLT

08049674	GOT[0]	0804969c	.dynamic section address
08049678	GOT[1]	400019f8	reloc entry address
0804967c	GOT[2]	4000596f	dynamic linker address
08049680	GOT[3]	0804845a	PLT[1] pushl address
08049684	GOT[4]	0804846a	PLT[2] pushl address

```
#PLT[0]: call dynamic linker
08048444: push *0x8049678 //*GOT[1]
0804844a: jmp *0x804967c //*GOT[2]

#PLT[1]  <printf>
08048454: jmp *0x8049680 //*GOT[3]
0804845a: pushl $0x0
0804845f:  jmp 0x8048444

#PLT[2]  <addvec>
08048464: jmp *0x8049684
0804846a: pushl $0x8
0804845f:  jmp 0x8048444
```