链接

Linking

本章内容

Topic

- 概念
  Concepts
- 符号解析
  Symbol Resolution
- 重定位
  Relocation
- 链接库
  Linking Library
- 库打桩技术
  Library Interpositioning

概念

Concepts

C程序示例
Example C Program

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

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

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

main.c

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

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

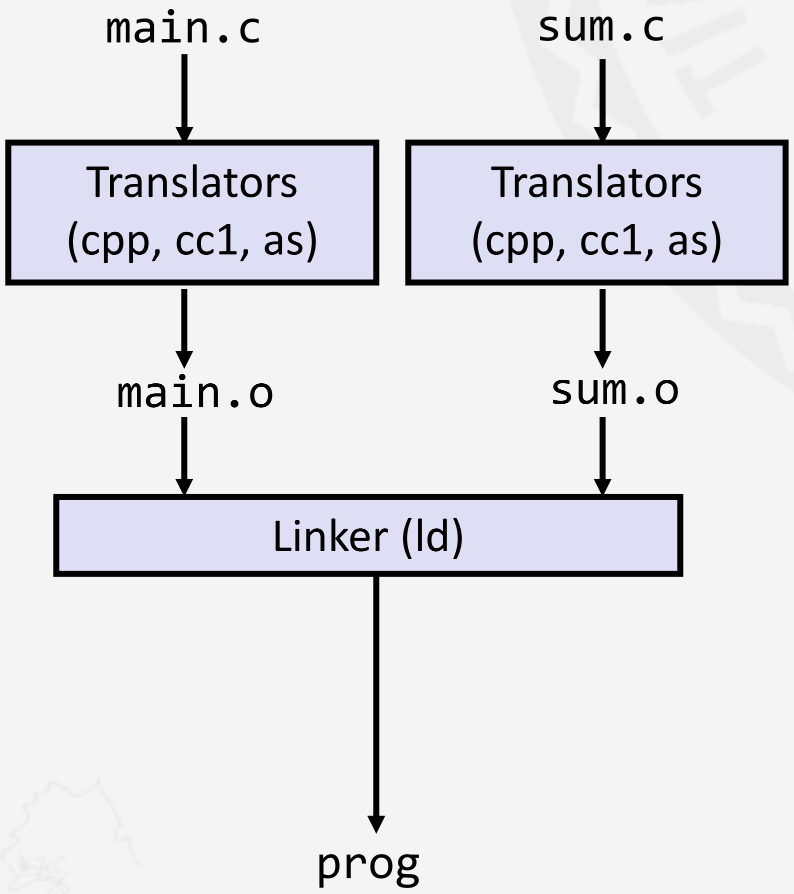
sum.c

概念

Concepts

静态链接
Static Linking

```
# 使用gcc驱动程序的编译和链接
# Programs are translated and linked using a
# compiler driver.
> gcc -Og -g -o prog main.c sum.c
> ./prog
```



源代码文件
Source files

独立的编译（并汇编）为<u>可重定位</u>目标文件
Separately compiled relocatable object files

完全链接的<u>可执行</u>目标文件（包括在main.c和sum.c中定义的所有指令和数据）
Fully linked <u>executable</u> object file (contains code and data for all functions defined in `main.c` and `sum.c`)

概念

Concepts

为什么需要链接
Why Linkers?

- **原因1：模块化**
  Reason 1: Modularity
  - 程序可以写成一个较小的源文件集合，而不是一个整体
    Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - 可以构建通用函数的库（稍后会有更多介绍）
    Can build libraries of common functions (more on this later)
    - 例如：数学库，标准C库
      e.g., Math library, standard C library

```
static
void func() {
  ……
}

int main()
{
  func();
  ……
  sum();
  ……
}
```
a.c

Module A

```
static
void func() {
  ……
}

int sum()
{
  …
  func();
  …
}
```
b.c

Module B

概念

Concepts

为什么需要链接
Why Linkers?

- **原因2：效率**
  Reason 2: Efficiency
  - **时间：** 独立编译
    Time: Separate compilation
    - 修改某一个源码文件后，只针对该文件进行编译，然后重新链接
      Change one source file, compile, and then relink
    - 不需要重新编译其他源码文件
      No need to recompile other source files
  - **空间：** 库
    Space: Libraries
    - 常用的函数可以聚合到单个文件中
      Common functions can be aggregated into a single file...
    - 而可执行文件和运行内存镜像中只包含实际使用的函数代码
      Yet executable files and running memory images contain only code for the functions they actually use

概念

Concepts

链接器都做些什么?
What Do Linkers Do?

- **第一步：符号解析**
  Step 1. Symbol resolution
  - 程序**定义**和**引用**符号（包括变量和函数）：
    Programs **define** and **reference** symbols (variables and functions):

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

  - 符号的定义都存储在（由编译器生成的）目标文件的**符号表**中
    Symbol definitions are stored in object file (by compiler) in **symbol table**
    - 符号表是一个结构体数组
      Symbol table is an array of structs
    - 每一项都包含着符号的名称、大小和位置等信息
      Each entry includes name, size, and location of symbol
  - 符号解析时，连接器将每个符号的引用与相应的符号定义进行关联
    During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.

概念

Concepts

链接器都做些什么?
What Do Linkers Do?

- **第二步：重定位**

  Step 2. Relocation

  - 将每个（目标文件中）对应独立的代码和数据节分别进行合并

    Merges separate code and data sections into single sections
  - 将.o文件中的符号的相对位置重定位到它们在可执行文件中的绝对内存位置

    Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable
  - 使用这些新的位置更新所有的符号引用

    Updates all references to these symbols to reflect their new positions

三种类型的目标文件（模块）
Three Kinds of Object Files (Modules)

- **可重定位目标文件　（.o文件）**
  Relocatable object file (.o file)
  - 包含某种形式的代码和数据，可与其他可重定位目标文件组合成可执行目标文件
    Contains code and data in a form that can be combined with other relocatable object files to form executable object file
  - 每个.o文件由对应的.c文件生成
    Each .o file is produced from exactly one source (.c) file

- **可执行目标文件（a.out文件）**
  Executable object file (a.out file)
  - 以可直接复制到内存中然后执行的形式包含代码和数据
    Contains code and data in a form that can be copied directly into memory and then executed

- **共享目标文件（.so文件）**
  Shared object file (.so file)
  - 特殊类型的可重定位目标文件，可以在加载时或运行时加载到内存并进行动态链接
    Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time
  - 在Windows上称为**动态链接库**　（.dll）
    Called **Dynamic Link Libraries** (DLLs) by Windows

概念

Concepts

可执行可连接的格式（ELF）
Executable and Linkable Format (ELF)

- 目标文件的标准二进制格式
  Standard binary format for object files
- 为以下文件提供一种统一的格式封装
  One unified format for
  - 可重定位目标文件（.o）
    Relocatable object files (.o)
  - 可执行目标文件（.o）
    Executable object files (a.out)
  - 共享目标文件（.o）
    Shared object files (.so)
- 通用名称：ELF二进制文件
  Generic name: ELF binaries

ELF目标文件格式
ELF Object File Format

- ELF头
  ELF header
  - 包含：字长、字节序、文件类型（.o, exec, .so）、机器类型等信息
    Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- 程序头部表（段头部表）
  Program header table (Segment header table)
  - 包含：页大小、虚拟地址内存段（节）、段大小的信息
    Page size, virtual addresses memory segments (sections), segment sizes
  - 可执行目标文件中必须存在此部分
    required for executables
- .text节
  .text section
  - 已编译程序的机器代码
    code

ELF header
Program header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table

概念

Concepts

ELF目标文件格式
ELF Object File Format

- .rodata节
  .rodata section
  - 只读数据：跳转表等
    Read only data: jump tables, …
- .data节
  .rodata section
  - 已初始化的全局变量
    Initialized global variables
- .bss节
  .bss section
  - 未初始化的全局变量
    Uninitialized global variables
  - 原名："块存储开始"
    Original name: Block Storage Start
  - 可以把它看做是"更好的节省空间"的缩写
    Take it as "Better Save Space"
  - 有节头信息，但不占空间
    Has section header but occupies no space

ELF header
Program header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table

概念

Concepts

ELF目标文件格式
ELF Object File Format

- **.symtab节**
  .symtab section
  - 符号表
    Symbol table
  - 存放定义和引用全局变量和函数的信息
    Defined and referenced global variables and procedures
- **.rel.text节**
  .rel.text section
  - .text节中的重定位信息
    Relocation info for .text section
  - 存放那些需要在可执行目标文件中被修改的指令地址信息
    Addresses of instructions that will need to be modified in the executable
  - 相应的指令需要在链接时被修改
    Instructions for modifying when linking

ELF header
Program header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table

概念

Concepts

ELF目标文件格式
ELF Object File Format

- **.rel.data节**
  .rel.data section
  - .data节中的重定位信息
    Relocation info for .data section
  - 在合并后的可执行文件中需要修改的数据所在的地址
    Addresses of pointer data that will need to be modified in the merged executable
- **.debug节**
  .debug section
  - 调试使用的符号信息（在gcc中使用-g选项生成）
    Info for symbolic debugging (gcc -g)
- **节头部表**
  Section header table
  - 每个节的偏移量和大小
    Offsets and sizes of each section

ELF header
Program header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table

本章内容

Topic

- 概念
  Concepts
- 符号解析
  Symbol Resolution
- 重定位
  Relocation
- 链接库
  Linking Library
- 库打桩技术
  Library Interpositioning

符号解析

Symbol Resolution

符号
Symbols

- **全局符号**
  Global symbols
  - 由模块m**定义**的并能被其他模块所引用
    Symbols **defined** by module m that can be referenced by other modules
    - 例如：**非static**的C函数和**非static**的全局变量
      E.g.: **non-static** C functions and **non-static** global variables
- **外部符号**
  External symbols
  - 由其他模块定义并被模块m中**引用**的全局符号
    Global symbols that are **referenced** by module m but defined by some other module
- **局部符号**
  Local symbols
  - 只能被模块m定义和引用的符号
    Symbols that are defined and referenced exclusively by module m
  - 例如：使用static关键字修饰的C函数和变量
    E.g.: C functions and variables defined with the static attribute
  - 局部符号不是局部变量
    Local linker symbols are not local program variables

符号解析

Symbol Resolution

解析符号
Resolving Symbols

定义
...that's defined here

引用
Referencing a global...

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

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

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

main.c

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

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

sum.c

定义
Defining
a global

链接器不知道
val的存在
Linker knows
nothing of `val`

引用
Referencing
a global...

定义
...that's defined here

链接器不知道 i 和 s
Linker knows
nothing of `i` or `s`

符号解析

Symbol Resolution

局部符号
Local Symbols

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

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

- 局部非static的C变量：存储在栈中
  Local non-static C variables: stored on the stack
- 局部static的C变量：存储在.bss或.data中
  Local static C variables: stored in either .bss or .data

编译器为每个x在.data节中分配空间
Compiler allocates space in `.data` for each definition of x

使用唯一的名称在符号表中创建局部符号，例如：x.1和x.2
Creates local symbols in the symbol table with unique names, e.g., `x.1` and `x.2`.

符号解析

Symbol Resolution

符号表结构
Symbol Structure

```
typedef struct {
    int  name;       /* String table offset */
    char type:4,     /* Data, func, section, or src file name */
         binding:4;  /* Local or global */
    char reserved;   /* Unused */
    short section;   /* Section header index, ABS, UNDEF or COMMON */
    long  value;     /* Section offset, or VM address */
    long  size;      /* Object size in bytes */
} Elf64_Symbol;
```

```
> readelf -a main.o
```

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

main.c

```
Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 array
     9: 0000000000000000    31 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sum
```

符号解析

Symbol Resolution

符号表结构
Symbol Structure

```
> readelf -a sum.o
```

```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 table '.symtab' contains 9 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS sum.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     8: 0000000000000000    69 FUNC    GLOBAL DEFAULT    1 sum
```

符号解析

Symbol Resolution

强符号和弱符号
Strong and Weak Symbols

- 全局符号包括强符号和弱符号
  Global symbols are either strong or weak
  - **强符号：** 函数和已初始化的全局变量
    Strong: procedures and initialized globals
  - **弱符号：** 未初始化的全局变量
    Weak: uninitialized globals

p1.c

```
int foo=5;     <- strong

p1()           <- strong
{
  ...
}
```

p2.c

```
int foo;       <- weak

p2()           <- strong
{
  ...
}
```

符号解析

Symbol Resolution

符号链接规则
Linker's Symbol Rules

- **在链接过程中，符号出现重名的情况时：**
  When symbols have the same name:

- **规则1：不允许有多个同名的强符号**
  Rule 1: Multiple strong symbols are not allowed
  - 每个强符号只能被定义一次，否则链接失败
    Each item can be defined only once, otherwise linking fails

- **规则2：如果有一个强符号和多个弱符号同名，那么选择强符号**
  Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol
  - 对弱符号的引用被解析为强符号
    References to the weak symbol resolve to the strong symbol

- **规则3：如果有多个弱符号同名，那么从这些弱符号中任意选择一个**
  Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - 可以通过`gcc -fno-common`选项重定义这一条规则（生成一个警告）
    Can override this with `gcc -fno-common (Generate a warning)`

解谜链接器
Linker Puzzles

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

```
p1() {}
```

链接错误：两个同名强符号 (p1)
Link time error: two strong symbols (**p1**)

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

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

引用x将指向相同的未初始化int型变量，这是你真正想要的结果吗？
References to **x** will refer to the same uninitialized int. Is this what you really want?

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

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

在p2中向x中写入值，可能会覆盖y！
Writes to `x` in **p2** might overwrite **y**!
讨厌的！
Evil!

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

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

在p2中向x写入值将会覆盖y
Writes to **x** in **p2** will overwrite y!
危险！
Nasty!

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

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

对x的引用都指向相同的已初始化变量
References to x will refer to the same initialized variable.

噩梦场景：两个相同的弱符号结构体，由不同的编译器使用不同的对齐规则进行编译。
Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

符号解析

Symbol Resolution

头文件的作用
Role of .h Files

```c
#ifdef INITIALIZE
  int g = 23;
  static int init = 1;
#else
  int g;
  static int init = 0;
#endif
```
global.h

```c
#include "global.h"

int f() {
  return g+1;
}
```
c1.c

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

int main() {
  if (!init)
    g = 37;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```
c2.c

符号解析

Symbol Resolution

执行预处理过程
Running Preprocessor

```c
#include "global.h"

int f() {
  return g+1;
}
```
c1.c

```c
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```
global.h

-DINITIALIZE

```c
int g = 23;
static int init = 1;
int f() {
  return g+1;
}
```

no initialization

```c
int g;
static int init = 0;
int f() {
  return g+1;
}
```

#include语句会使C预处理器将引用的文件一字不差的插入到当前文件中（可使用gcc -E查看结果）

#include causes C preprocessor to insert file verbatim (Use gcc –E to view result)

使用全局变量
Using Global Variables

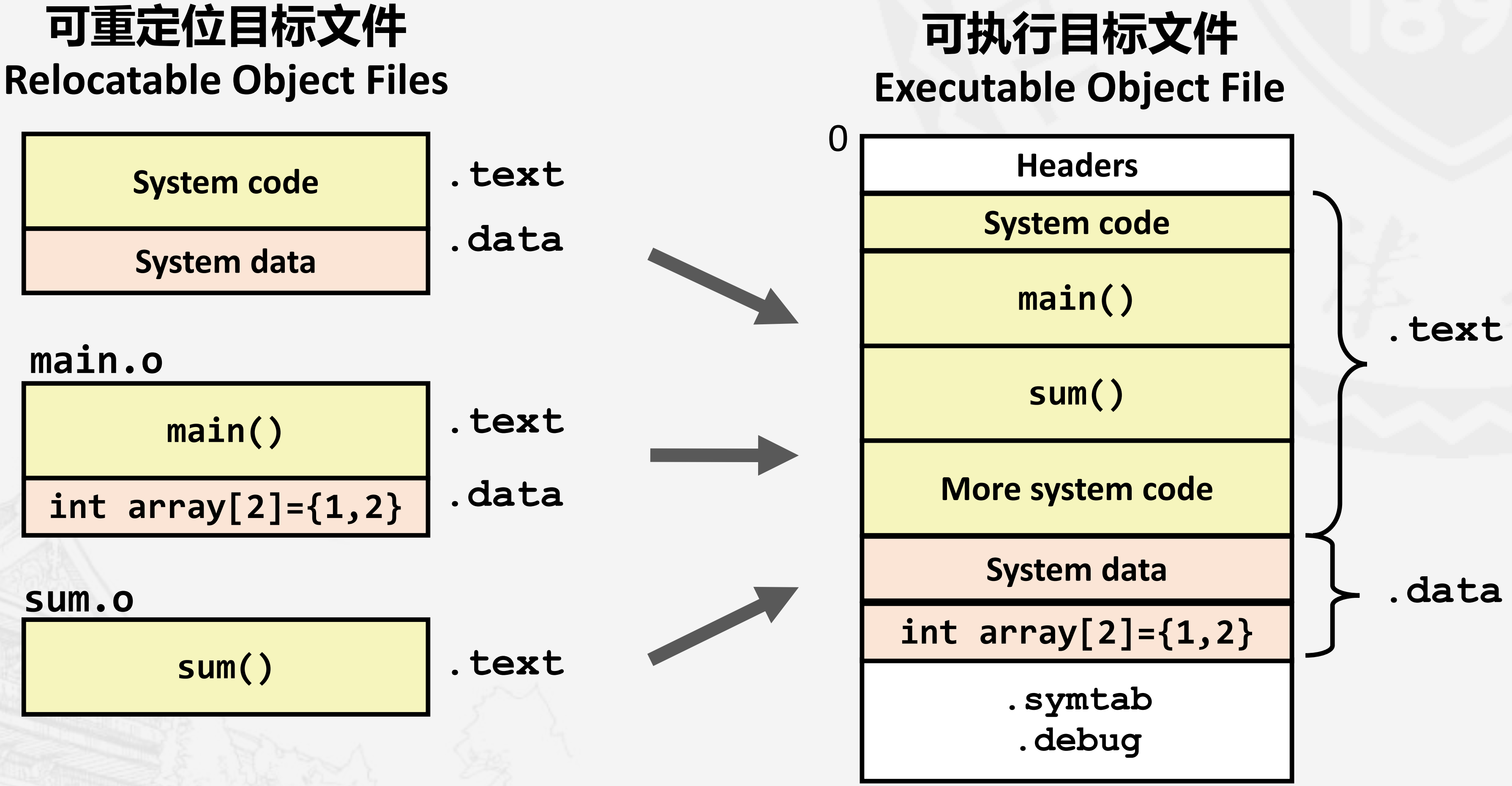
- **尽可能避免使用全局变量**
  Avoid if you can
- **否则**
  Otherwise
  - 如果可能的话，使用`static`关键字修饰它
    Use `static` if you can
  - 如果定义一个全局变量，初始化它
    Initialize if you define a global variable
  - 如果你在当前模块中使用外部模块定义的全局变量，请在当前模块中使用`extern`关键字进行声明
    Use `extern` if you use external global variable

本章内容

Topic

- 概念
  Concepts
- 符号解析
  Symbol Resolution
- 重定位
  Relocation
- 链接库
  Linking Library
- 库打桩技术
  Library Interpositioning

重新定位代码和数据
Using Global Variables



重定位

Relocation

重定位信息数据结构
Relocation Item Structure

```
typedef struct {
    long offset;        /* Offset of the reference to relocate */
    long type: 32,      /* Relocation type */
         symbol: 32;    /* Symbol the reference should point to */
    long addend;        /* Constant part of relocation expression */
} Elf64_Rela;
```

0	64	96	128
offset	type	symbol	addend

- **重定位类型**
  Relocation type
  - R_X86_64_PC32 : 相对引用
    Relative Reference
  - R_X86_64_32 : 绝对引用
    Absolute Reference

重定位

Relocation

.text节的重定位
Relocation Info in .text

```
> objdump -r -d main.o
```

main.c

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

main.o

```
Disassembly of section .text:

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
                        a: R_X86_64_32  array
   e:   e8 00 00 00 00          callq  13 <main+0x13>
                        f: R_X86_64_PC32        sum-0x4
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

```
r.offset = 0xf
r.symbol = sum
r.type   = R_X86_64_PC32
r.addend = -0x4
```

重定位

Relocation

相对引用的重定位
Relative Reference in Relocation

- **If**
  - ADDR(s) = ADDR(.text) = 0x4004d0
  - ADDR(r.symbol) = ADDR(sum) = 0x4004e8
- **Then**
  - refaddr = ADDR(s) + r.offset
    = 0x4004d0 + 0xf
    = 0x4004df
  - refptr = s + r.offset
  - *refptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr)
    = (unsigned)(0x4004e8 + (-4) – 0x04004df)
    = (unsigned)(0x5)

```
Disassembly of section .text:

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
                    a: R_X86_64_32  array
   e:   e8 00 00 00 00         callq  13 <main+0x13>
                    f: R_X86_64_PC32      sum-0x4
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

重定位后
After Relocation

```
00000000004004d0 <main>:
  4004d0:      48 83 ec 08         sub    $0x8,%rsp
  4004d4:      be 02 00 00 00      mov    $0x2,%esi
  4004d9:      bf 30 10 60 00      mov    $0x601030,%edi
  4004de:      e8 05 00 00 00      callq  4004e8 <sum>
  4004e3:      48 83 c4 08         add    $0x8,%rsp
  4004e7:      c3                  retq
```

重定位

Relocation

.data节的重定位
Relocation Info in .data

main.c

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

```
> objdump -r -d -j .data main.o
```

```
Disassembly of section .data:

0000000000000000 <array>:
   0:   01 00 00 00 02 00 00 00
```

重定位

Relocation

绝对引用的重定位
Absolute Reference in Relocation

main.c

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

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

main.o

```
Disassembly of section .text:

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
                        a: R_X86_64_32  array
   e:   e8 00 00 00 00          callq  13 <main+0x13>
                        f: R_X86_64_PC32        sum-0x4
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

- If
  - ADDR(s) = ADDR(.text) = 0x4004d0
  - ADDR(r.symbol) = ADDR(array) = 0x601018
- Then
  - refptr = s + r.offset
  - *refptr = (unsigned)(ADDR(r.symbol) + r.addend) = (unsigned)(0x601018 + 0) = (unsigned)(0x601018)

重定位

Relocation

重定位后的情况（.text）
After Relocation(.text)

```
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
  4004de:       e8 05 00 00 00          callq  4004e8 <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
```

sum()使用相对PC的地址：

Using PC-relative addressing for sum():

0x4004e8 = 0x4004e3 + 0x5

加载目标文件
Loading Executable Object Files

可执行目标文件
Executable Object File

0
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)

Address	Memory region	Note
	内核虚拟内存 Kernel virtual memory	此部分内存对用户代码不可见 Memory invisible to user code
	用户栈（运行时创建）User stack (created at runtime)	`%rsp` (栈指针) (stack pointer)
	↓ ↑	
	共享库内存映射区 Memory-mapped region for shared libraries	
	↑	`brk`
	运行时堆（由malloc创建）Run-time heap (created by `malloc`)	
	可读写的段 Read/write segment (`.data`, `.bss`)	从可执行目标文件中加载 Loaded from the executable file
`0x400000`	只读段 Read-only segment (`.init`, `.text`, `.rodata`)	
0	未使用 Unused	

本章内容

Topic

- 概念
  Concepts
- 符号解析
  Symbol Resolution
- 重定位
  Relocation
- 链接库
  Linking Library
- 库打桩技术
  Library Interpositioning

打包常用的函数
Packaging Commonly Used Functions

- 如何打包程序员的常用函数
  How to package functions commonly used by programmers?
  - 数学、输入输出、内存管理、字符串操作等
    Math, I/O, memory management, string manipulation, etc
- 考虑到目前为止给大家介绍的链接器架构，这个问题的解决很尴尬：
  Awkward, given the linker framework so far:

- **选项1：** 把所有的函数代码放到一个文件中
  Option 1: Put all functions into a single source file
  - 程序员需要链接一个大的目标文件到他们的程序中
    Programmers link big object file into their programs
  - 无论是时间效率还是空间效率都很差
    Space and time inefficient

- **选项2：** 把每一个函数作为一个独立的代码文件
  Option 2: Put each function in a separate source file
  - 程序员显式地指定相应的目标文件链接到程序中
    Programmers explicitly link appropriate binaries into their programs
  - 更有效率，但是对于程序员来说太麻烦了
    More efficient, but burdensome on the programmer

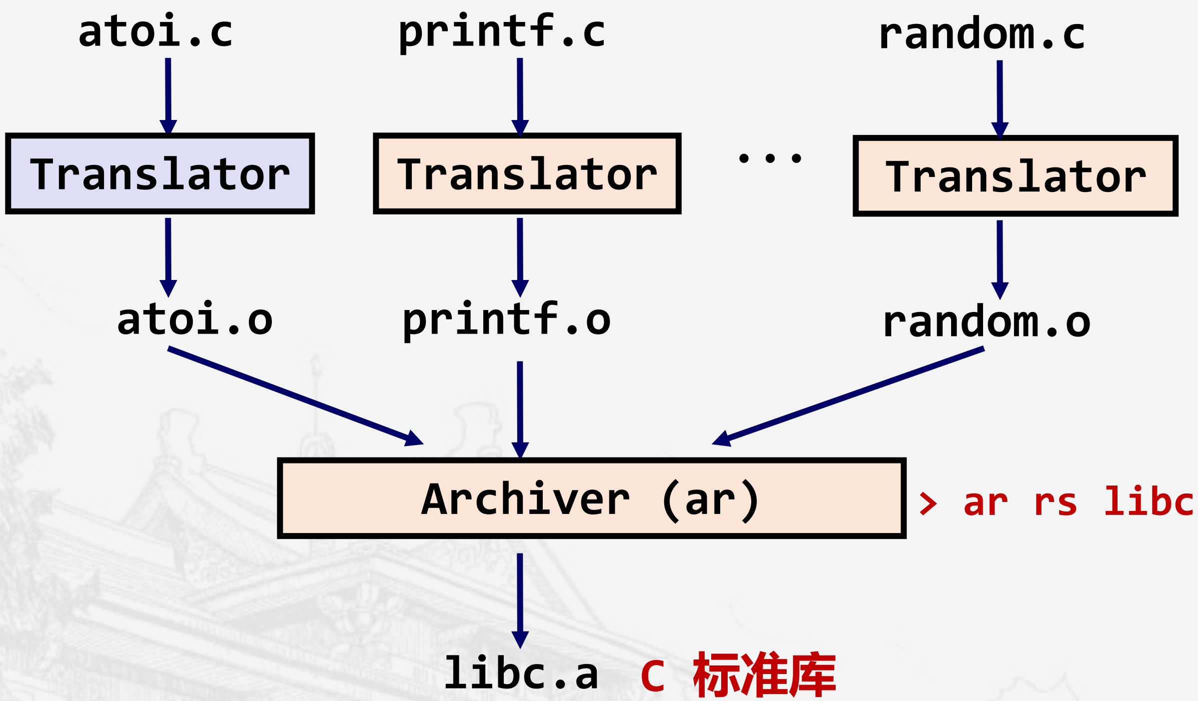
解决方案：静态库
Solution: Static Libraries

- 静态库（.a 归档文件）
  Static libraries (.a archive files)
  - 将相关的可重定位对象文件集成到一个带有索引的文件中（这个文件称为归档文件）
    Concatenate related relocatable object files into a single file with an index (called an archive)
  - 增强链接器的功能，使其可以通过在一个或多个归档文件中查找符号，来尝试解析哪些未被解析的外部符号引用
    Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives
  - 如果归档文件中的某个成员（可重定位目标文件）被引用，则将其链接到可执行目标文件中
    If an archive member file resolves reference, link it into the executable

链接库

Linking Library

创建静态库
Creating Static Libraries



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

C 标准库

C standard library

归档器允许增量更新

Archiver allows incremental updates

重新编译函数，改变和替换归档中相应的.o文件

Recompile function that changes and replace .o file in archive

链接库

Linking Library

常用库
Commonly Used Libraries

- libc.a（C标准库）
  libc.a (the C standard library)
  - 4.6MB的归档文件，包含1496个目标文件
    4.6 MB archive of 1496 object files
  - 包含：输入输出、内存分配、信号处理、字符串处理、日期和时间、随机数、整数数学运算等
    I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- libm.a（C数学库）
  libm.a (the C math library)
  - 2MB的归档文件，包含444个目标文件
    2 MB archive of 444 object files.
  - 浮点数数学运算（sin、cos、tan、log、exp、sqrt等）
    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
…
```

链接库

Linking Library

使用静态库进行链接
Linking with Static Libraries

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

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

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

main2.c

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

addvec.c

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

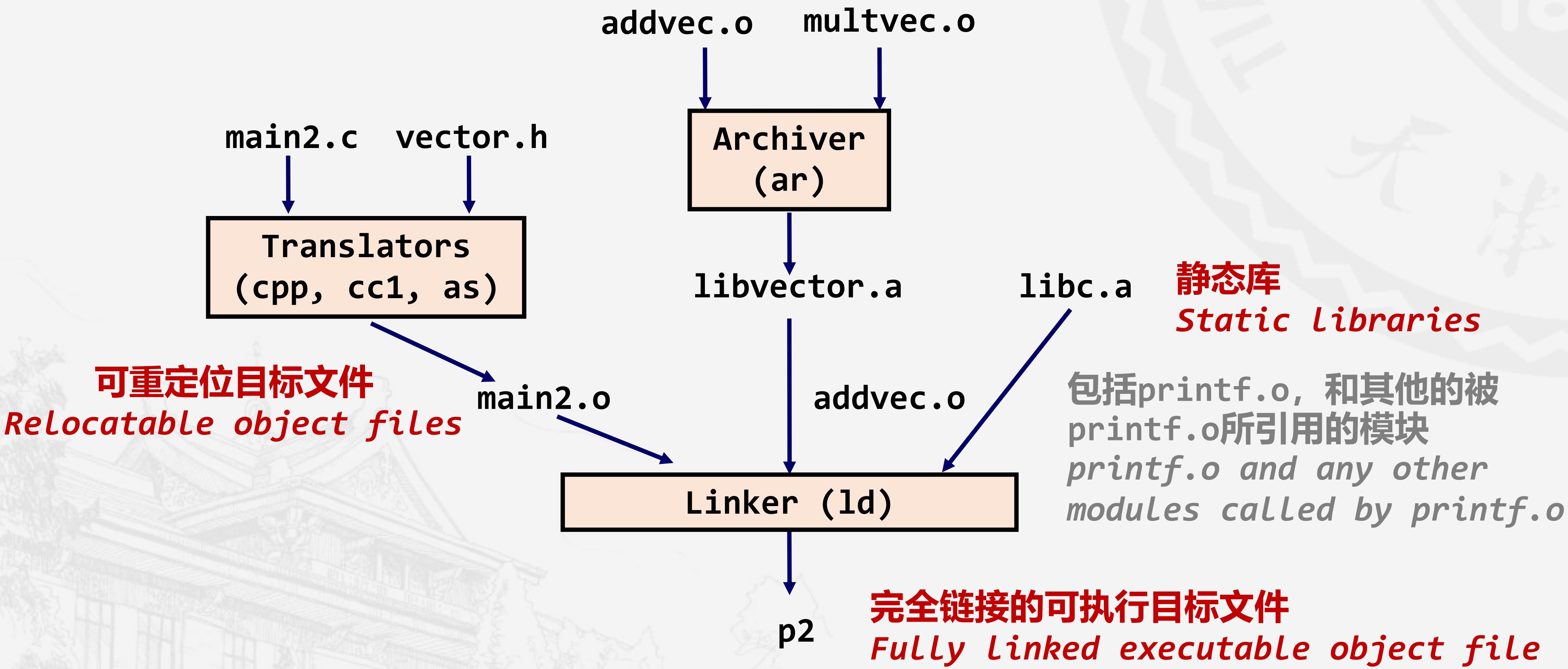
multvec.c

libvector.a

链接库

Linking Library

使用静态库进行链接
Linking with Static Libraries



使用静态库
Using Static Libraries

- 链接器解析外部符号的算法
  Linker's algorithm for resolving external references:
  - 按照命令行上出现的顺序依次扫描.o和.a文件
    Scan .o files and .a files in the command line order
  - 在扫描过程中，维护一个当前未解析引用的列表
    During the scan, keep a list of the current unresolved references
  - 当遇到每个新的.o或.a文件中对象所定义的符号时，尝试根据这些符号解析列表中未解析的引用
    As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in ob
  - 如果扫描结束后，未解析列表不为空，则报错
    If any entries in the unresolved list at end of scan, then error

链接库

Linking Library

使用静态库
Using Static Libraries

- 可能出现的问题:
  Problem:
  - 命令行中的顺序十分重要!
    Command line order matters!
  - 常规解决方案：可以把库放在命令行的最后
    Moral: put libraries at the end of the command line.

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

> gcc -L. libtest.o -lmine    √
```

链接库

Linking Library

共享库
Shared Libraries

- 静态库有以下缺点：
  Static libraries have the following disadvantages:
  - 在可执行目标文件中重复出现（每个函数几乎都需要C标准库）
    Duplication in the stored executables (every function need std libc)
  - 在运行可执行文件时，在内存中重复出现
    Duplication in the running executables
  - 系统库中的小bug修复后，需要为每个应用程序显式的进行重新链接
    Minor bug fixes of system libraries require each application to explicitly relink

- 现代的解决方案：共享库
  Modern solution: Shared Libraries
  - 包含代码和数据的对象文件，可以在程序加载时或运行时时动态的加载并连接
    Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
  - 也被称为：动态链接库、`DLL`、`.so`文件
    Also called: dynamic link libraries, DLLs, .so files

共享库
Shared Libraries

- 动态链接可以在可执行程序第一次加载运行时发生（加载期链接）：
  Dynamic linking can occur when executable is first loaded and run (load-time linking):
  - Linux中，由动态链接器（ld-linux.so）进行自动管理
    Common case for Linux, handled automatically by the dynamic linker (ld-linux.so).
  - C标准库通常采用动态链接
    Standard C library (libc.so) usually dynamically linked

- 动态链接也可以放生在程序运行后（运行期链接）
  Dynamic linking can also occur after program has begun (run-time linking)
  - 在Linux中，通过调用dlopen()接口实现，可以用于:
    In Linux, this is done by calls to the dlopen() interface
  - 软件的分发
    Distributing software
  - 高性能Web服务器
    High-performance web servers
  - 运行时库打桩
    Runtime library interpositioning

- 共享库例程可以由多个进程共享（“虚拟内存”一章中会讲到）
  Shared library routines can be shared by multiple processes (More on this when we learn about virtual memory)

加载期动态链接
Dynamic Linking at Load-time

main2.c　vector.h

↓

Translators
(cpp, cc1, as)

↓

main2.o

可重定位目标文件
Relocatable object file

```
> gcc -shared -fPIC -o libvector.so addvec.c multvec.c
```

libc.so
libvector.so

重定位和符号表信息
Relocation and symbol table info

↓

Linker (ld)

↓

p2

部分链接的可执行目标文件
Partially linked executable object file

↓

加载器
Loader
(execve)

libc.so
libvector.so

代码和数据
Code and data

↓

动态链接器
Dynamic linker (ld-linux.so)

在内存中完全链接的可执行程序
Fully linked executable in memory

运行期动态链接
Dynamic Linking at Run-time

```c
#include <stdio.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 lib that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    ......
```

运行期动态链接
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;
}
```

位置无关的代码（PIC）
Position-Independent Code (PIC)

- 在gcc中，使用编译选项 `-fPIC`
  Use options `-fPIC` in gcc
  - `gcc -shared -fPIC -o libvector.so addvec.c multvec.c`
- PIC中的数据引用
  PIC Data References
  - 使用**全局偏移量表**（GOT）来引用全局变量
    Using the **global offset table** (GOT) to reference a global variable

```
.data
GOT[0]: …
GOT[1]: …
GOT[2]: …
GOT[3]: &addcnt
```

```
.text
addvec:
    mov 0x2008b9(%rip), %rax     # %rax=*GOT[3]=&addcnt
    addl $0x1,(%rax)             # addcnt++
```

在运行时，GOT[3]和addl指令间的固定距离是0x2008b9

Fixed distance of 0x2008b9 bytes at run time between GOT[3] and addl instruction

位置无关的代码（PIC）
Position-Independent Code (PIC)

- PIC中的函数调用
  PIC Function Calls
  - 使用过程链接表（PLT）和**全局偏移量表**（GOT）实现外部函数调用
    Using the **procedure linkage table** (PLT) and GOT to call external functions

```
.data
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()
        overwrite
GOT[5]: 0x4005d6 # printf()
```

addvec的第一次调用
First invocation of *addvec*

```
.text
        callq 0x4005c0     # call addvec()
......
# Procedure linkage table (PLT)
# PLT[0]: call dynamic linker
4005a0: pushq  *GOT[1]
4005a6: jmpq   *GOT[2]        ④
......            ③       ①
# PLT[2]: call addvec()
4005c0: jmpq   *GOT[4]
                ②
4005c6: pushq  $0x1      # addvec ID = 0x1
4005cb: jmpq   4005a0
```

位置无关的代码（PIC）
Position-Independent Code (PIC)

- PIC中的函数调用
  PIC Function Calls
  - 使用过程链接表（PLT）和**全局偏移量表**（GOT）实现外部函数调用
    Using the **procedure linkage table** (PLT) and GOT to call external functions

```
.data
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 # addvec()
GOT[5]: 0x4005d6 # printf()
```

addvec的后续调用
Subsequent invocation of *addvec*

```
.text
        callq 0x4005c0     # call addvec()
……
# Procedure linkage table (PLT)
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
……
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1     # addvec ID = 0x1
4005cb: jmpq  4005a0
```

①
②

总结
Summary

- 链接是一种允许程序通过多个目标文件构建的技术
  Linking is a technique that allows programs to be constructed from multiple object files

- 链接可能发生在一个程序生命周期的不同阶段：
  Linking can happen at different times in a program’s lifetime:
  - 编译期（当一个程序被编译时）
    Compile time (when a program is compiled)
  - 加载期（当一个程序加载进入内存时）
    Load time (when a program is loaded into memory)
  - 运行期（当一个程序执行时）
    Run time (while a program is executing)

- 理解链接可以帮助你规避哪些讨厌的错误，使你成为一名更优秀的程序员
  Understanding linking can help you avoid nasty errors and make you a better programmer

本章内容

Topic

- 概念
  Concepts
- 符号解析
  Symbol Resolution
- 重定位
  Relocation
- 链接库
  Linking Library
- 库打桩技术
  Library Interpositioning

库打桩技术

Library Interpositioning

案例研究
Case Study

- 库打桩：一种强大的链接技术，允许程序员对任意的函数调用进行拦截
  Library interpositioning : powerful linking technique that allows programmers to intercept calls to arbitrary functions

- 库打桩技术可以发生在：
  Interpositioning can occur at:
  - 编译期：当源代码被编译时
    Compile time: When the source code is compiled
  - 链接期：当可重定位目标文件以静态链接的方式生成可执行目标文件时
    Link time: When the relocatable object files are statically linked to form an executable object file
  - 加载/运行期：当一个可执行目标文件加载进入内存时，动态链接，然后执行
    Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed

库打桩技术

Library Interpositioning

应用场景
Applications

- 安全
  Security
  - 约束（沙箱）
    Confinement (sandboxing)
    - 拦截对`libc`的函数调用
      Interpose calls to libc functions
  - 背景加密
    Behind the scenes encryption
    - 自动地加密那些未加密的网络连接
      Automatically encrypt otherwise unencrypted network connections

- 检测和分析
  Monitoring and Profiling
  - 对函数调用进行计数
    Count number of calls to functions
  - 分析函数的调用次序和参数
    Characterize call sites and arguments to functions
  - 追踪`malloc`调用
    malloc tracing
    - 侦测内存泄漏
      Detecting memory leaks
    - 生成地址轨迹
      Generating address traces

库打桩技术

Library Interpositioning

示例程序
Example program

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

int main()
{
    int *p = (int *)malloc(32);
    free(p);
    printf("hello, world\n");
    exit(0);
}
```
hello.c

- 目标：在不修改源代码的前提下，跟踪分配和释放的块的地址和大小
  Goal: trace the addresses and sizes of the allocated and freed blocks, without modifying the source code
- 三种解决方案：在编译期、链接期和加载/运行期对malloc和free库函数进行“打桩”。
  Three solutions: interpose on the lib malloc and free functions at compile time, link time, and load/run time

编译期打桩
Compile-time Interpositioning

```
#ifdef COMPILETIME
/* Compile-time interposition of malloc and free using C
 * preprocessor. A local malloc.h file defines malloc (free)
 * as wrappers mymalloc (myfree) respectively.
 */

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

/*
 * mymalloc - malloc wrapper function
 */
void *mymalloc(size_t size, char *file, int line)
{
    void *ptr = malloc(size);
    printf("%s:%d: malloc(%d)=%p\n", file, line, (int)size, ptr);
    return ptr;
}
#endif
```

mymalloc.c

编译期打桩
Compile-time Interpositioning

```
#define malloc(size) mymalloc(size, __FILE__, __LINE__ )
#define free(ptr) myfree(ptr, __FILE__, __LINE__ )

void *mymalloc(size_t size, char *file, int line);
void myfree(void *ptr, char *file, int line);
```

malloc.h

```
linux> make helloc
gcc -O2 -Wall -DCOMPILETIME -c mymalloc.c
gcc -O2 -Wall -I. -o helloc hello.c mymalloc.o
linux> make runc
./helloc
hello.c:7: malloc(10)=0x501010
hello.c:7: free(0x501010)
hello, world
```

库打桩技术

Library Interpositioning

链接期打桩
Link-time Interpositioning

```c
#ifdef LINKTIME
/*
   Link-time interposition of malloc and free using the static
   linker's (ld) "--wrap symbol" flag.
*/

#include <stdio.h>

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

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

mymalloc.c

链接期打桩
Link-time Interpositioning

```
linux> make hellol
gcc -O2 -Wall -DLINKTIME -c mymalloc.c
gcc -O2 -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o hellol hello.c mymalloc.o
linux> make runl
./hellol
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- 使用编译选项 -Wl，表示把参数传递给链接器
  The “-Wl” flag passes argument to linker
- --wrap,malloc用来通知链接器使用特殊的方法进行符号解析
  Telling linker “--wrap,malloc ” tells it to resolve references in a special way:
  - 对malloc符号的引用被解析为__wrap_malloc符号
    Refs to malloc should be resolved as __wrap_malloc
  - 对__real_malloc符号的引用被解析为malloc符号
    Refs to __real_malloc should be resolved as malloc

库打桩技术

Library Interpositioning

加载/运行期打桩
Load/Run-time Interpositioning

```c
#ifdef RUNTIME
 /* Run-time interposition of malloc and free based on
  * dynamic linker's (ld-linux.so) LD_PRELOAD mechanism */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

void *malloc(size_t size)
{
    static void *(*mallocp)(size_t size);
    char *error;
    void *ptr;

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

mymalloc.c

库打桩技术

Library Interpositioning

加载/运行期打桩
Load/Run-time Interpositioning

```
linux> make hellor
gcc -O2 -Wall -DRUNTIME -shared -fPIC -o mymalloc.so mymalloc.c
gcc -O2 -Wall -o hellor hello.c
linux> make runr
(LD_PRELOAD="/usr/lib64/libdl.so ./mymalloc.so" ./hellor)
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- 环境变量 LD_PRELOAD，用来通知动态链接器在解析（尚未被解析的）符号时（例如：malloc），首先对libdl.so和mymalloc.so进行扫描
  The LD_PRELOAD environment variable tells the dynamic linker to resolve unresolved refs (e.g., to malloc)by looking in libdl.so and mymalloc.so first.
  - libdl.so用于解析dlopen符号
    libdl.so necessary to resolve references to the dlopen functions

小结
Summary

- 编译期
  Compile Time
  - 对`malloc/free`的调用被宏扩展成对`mymalloc/myfree`的调用
    Apparent calls to malloc/free get macro-expanded into calls to mymalloc/myfree
- 链接期
  Link Time
  - 使用链接器的技巧实现对特殊名称符号的解析
    Use linker trick to have special name resolutions
    - `malloc → __wrap_malloc`
    - `__real_malloc → malloc`
- 加载/运行期
  Load/Run Time
  - 使用动态链接器加载实现自定义版本的`malloc/free`
    Implement custom version of malloc/free that use dynamic linking to load library malloc/free under different names