清華大學
Tsinghua University

2024秋季

计算机系统概论

Introduction to Computer Systems

80X86汇编语言与C语言-5

韩文弢

hanwentao@tsinghua.edu.cn

计算机体系结构

C程序在硬件层面的表示

- 数据/代码的内存地址定位
- 链接
- 数据/代码的内存布局
- ……

main.c

```
int array[4] = {1,2,3,4};
static int brray[4] = {1,2,3,4};

static int intra_sum (int x[4],int y)
{
   return x[y-1];
}
int main()
{
    int val = intra_sum(array, 3) + inter_sum(brray,3);
    return val;
}
```

编译

```
intra_sum:
        movslq  %esi, %rsi
        movl    -4(%rdi,%rsi,4), %eax
        ret
main:
        pushq   %rbx
        movl    $3, %esi
        movl    $array, %edi
        call    intra_sum
        movl    %eax, %ebx// val -> ebx
        movl    $3, %esi
        movl    $brray, %edi
        movl    $0, %eax
        call    inter_sum
        addl    %ebx, %eax// val -> eax
        popq    %rbx
        ret
brray:
        .long   1
        .long   2
        .long   3
        .long   4
array:
        .long   1
        .long   2
        .long   3
        .long   4
```

汇编指令

main.o

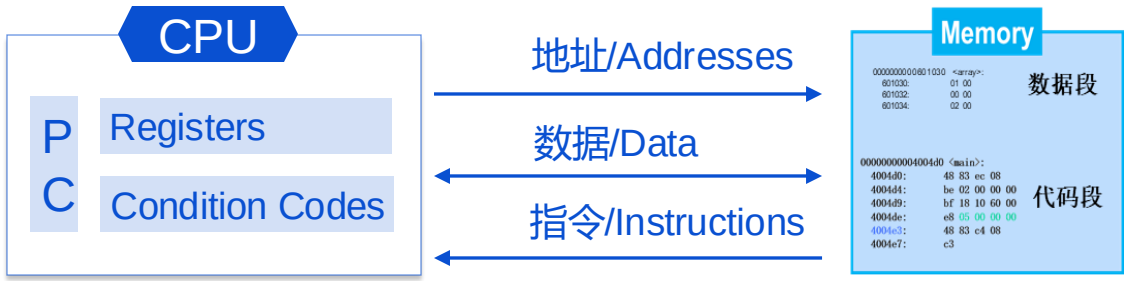
链接

```
00000000004004de <main>:
  4004de:       53                      push   %rbx
  4004df:       be 03 00 00 00          mov    $0x3,%esi
  4004e4:       bf 40 10 60 00          mov    $0x601040,%edi
  4004e9:       e8 e8 ff ff ff          callq  4004d6 <intra_sum>
  4004ee:       89 c3                   mov    %eax,%ebx
  4004f0:       be 03 00 00 00          mov    $0x3,%esi
  4004f5:       bf 30 10 60 00          mov    $0x601030,%edi
  4004fa:       b8 00 00 00 00          mov    $0x0,%eax
  4004ff:       e8 04 00 00 00          callq  400508 <inter_sum>
  400504:       01 d8                   add    %ebx,%eax
  400506:       5b                      pop    %rbx
  400507:       c3                      retq
```

仅给出部分内容

内存地址

运行



程序/数据在硬件层面的表示与运行

C语言示例程序

```
int array[4] = {1,2,3,4};
static int brray[4] = {1,2,3,4};

static int intra_sum (int x[4],int y)
{
   return x[y-1];
}

int main()
{
    int val = intra_sum(array, 3) +
              inter_sum(brray,3);
    return val;
}
```

main.c

```
int inter_sum (int x[4],int y)
{
   return x[y-1];
}
```

sum.c

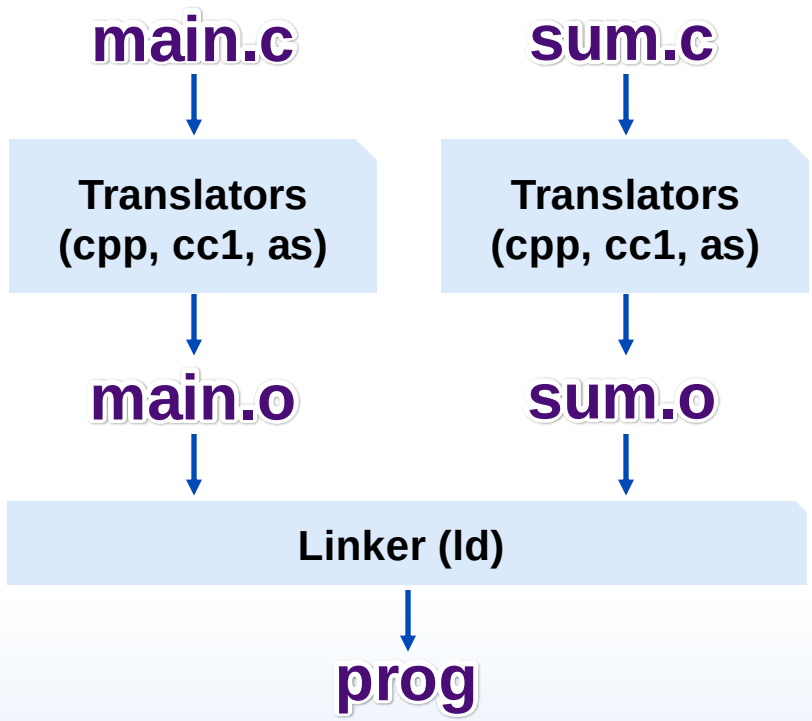
迄今为止的数据、函数、指令“定位”

	编译	链接（静态）	说明
局部变量	√		存于栈内 or 寄存器 or 被优化掉
全局变量?			**各个模块(.o文件)的数据段、代码段合并后，方能确定绝对地址**
函数参数	√		通过指定的寄存器与栈传递
函数返回值	√		%rdx + %rax
跳转指令（直接）	√		相对于PC的offset
函数返回地址	√		存于栈顶
静态函数入口地址	√		相对于PC的offset
全局函数入口地址?			**各个模块(.o文件)的数据段、代码段合并后，方能确定绝对地址**

静态链接(Static Linking)

编译与链接

- unix> gcc -Og -o prog main.c sum.c *-static -fno-pie*
- unix> ./prog



源文件

编译生成的可重定位对象文件（relocatable object file）

完全链接后产生的可执行对象文件（executable object file）
（包含有`main.c`与`sum.c`中的所有代码与数据 ）

程序链接的作用 1: 模块化好

- 多个小的源文件可以组成（链接成）一个程序，而不是一个巨大的单一源文件
- 可以将多个通用函数链接成库文件

.....

数学计算库

标准C库

作用 2: 工作效率高

省时间: 独立编译

- 某个原文件被修改后，可以独立编译并重链接
- 而不需要编译所有文件

省空间: 库文件

- 多个通用函数可以被集成到一个文件中
- 可执行文件及其运行时的内存镜像（memory image）内只包含有实际使用到的函数

链接步骤 1. 符号解析

程序定义以及引用了一系列符号（symbols，包括变量与函数):

- void swap() {…} /* 定义 *swap* */
- swap(); /* 引用 *swap* */
- int *xp = &x; /* 定义*xp*, 引用 *x* */

编译器将符号定义存储在符号表（symbol table）中

- 符号表是一个结构体数组
- 每个条目包括符号的名称、大小和位置

链接器将每一个符号引用（reference）与符号定义联系起来

链接步骤 2. 重定位

将多个文件的数据 / 代码段集成为单一的数据段和代码段

将.o文件中的符号解析为绝对地址

然后将所有的符号引用更新为这些新的地址

三种不同的对象文件

重定向对象文件 (.o 文件)

- 含有一定格式的代码与数据内容，可以与其他重定向对象文件一起集成为执行文件
  - 一个.o 文件由唯一的一个源文件生成

执行文件(a.out 文件)

- 含有一定格式的代码与数据内容，可以直接被装载入内存并执行

共享对象文件 (.so 文件)

- 特殊类型的重定向对象文件，可以被装载入内存后进行动态链接；链接可以在装载时或者运行时完成
- Windows系统下被称为DLL文件

Executable and Linkable Format (ELF)

对象文件的标准二进制格式（之一）

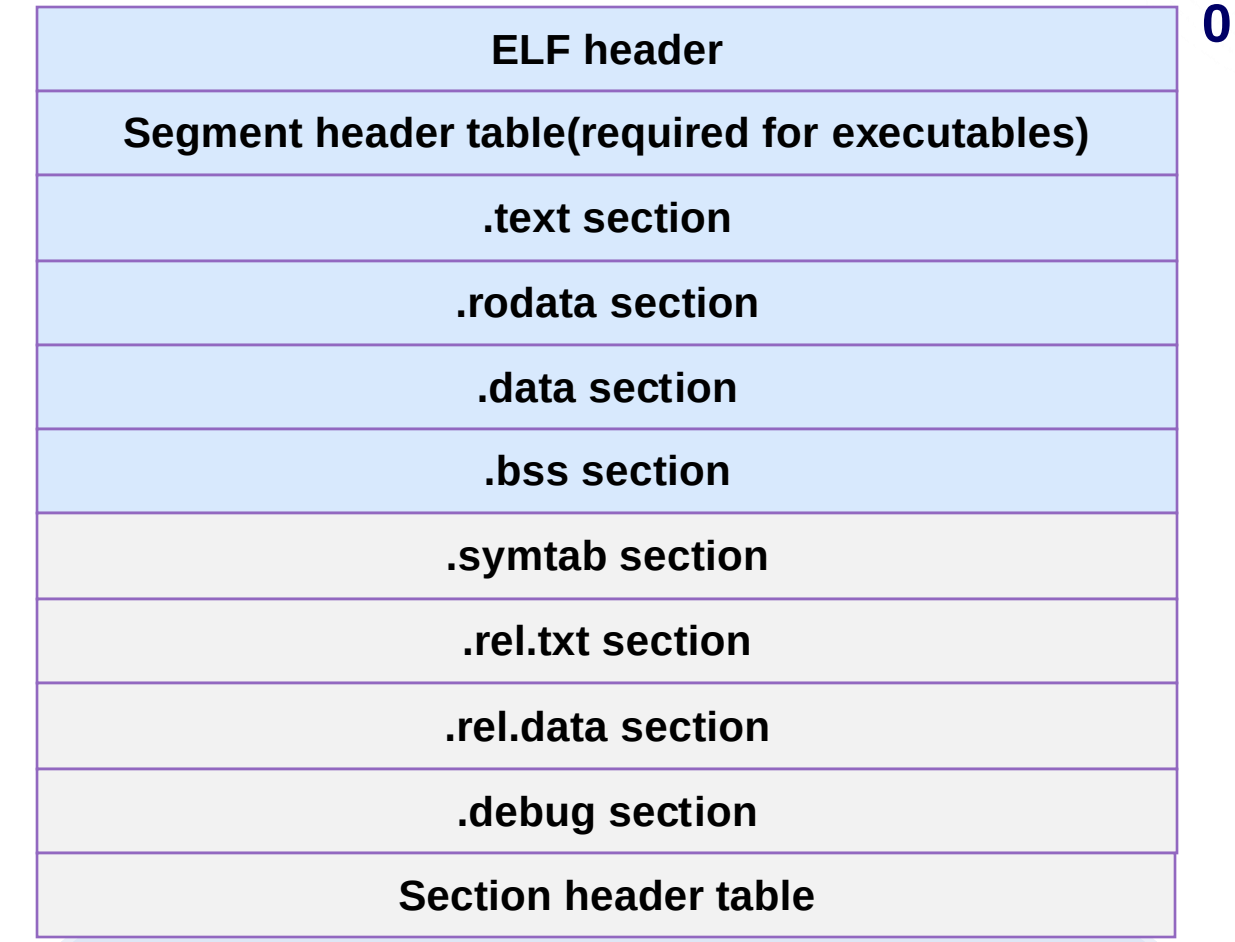
上面提到的三种文件都可以采用这一统一格式

最初由AT&T System V Unix系统采用

后来被广泛采用，包括BSD Unix与Linux系统

ELF 文件的格式

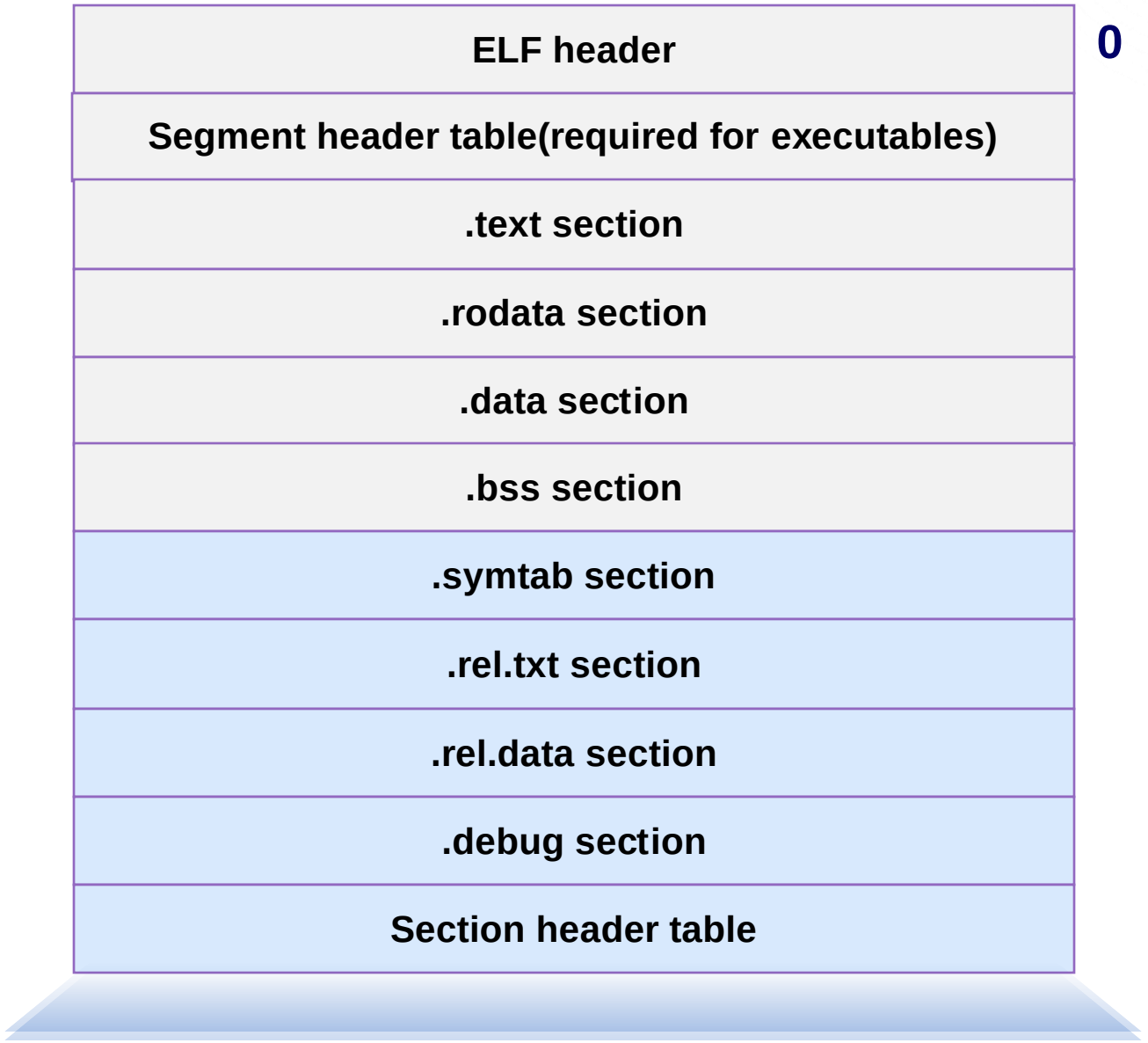
- **ELF header**
  - 机器字长度,字节序,文件类型(.o, exec, .so), 机器类型等
- **Segment header table**
  - 页大小, 虚拟地址内存段(节)，段大小等
- **.text section**
  - 代码
- **.rodata section**
  - 只读数据，跳转表， …
- **.data section**
  - 初始化的全局变量
- **.bss section**
  - 未初始化的全局变量*
  - 能够更好的节省空间
  - 因为其不在文件中占据实际空间



*初始化为0的global variable 被GCC 放到 .bss 中——C语言规范规定未初始化的全局变量/局部静态变量需要自动初始化为0

ELF 文件的格式(续前)

- **.symtab section**
  - 符号表
  - 全局过程与变量的名字
    段名称和位置
- **.rel.text section**
  - .text段的重定位信息
  - 在可执行文件中需要修改的指令的地址
    以及相应的修改指令
- **.rel.data section**
  - .data段的重定位信息
  - 在可执行文件中需要修改的指针数据的地址
- **.debug section**
  - 调试时的符号信息 (gcc -g)
- **Section header table**
  - 每个 section 的偏移量和大小



链接符号(.symtab section)

全局符号

- 某一个模块定义的、且可以被其他模块引用的变量或者函数符号
- 比如非静态(non-static)函数以及非静态全局变量

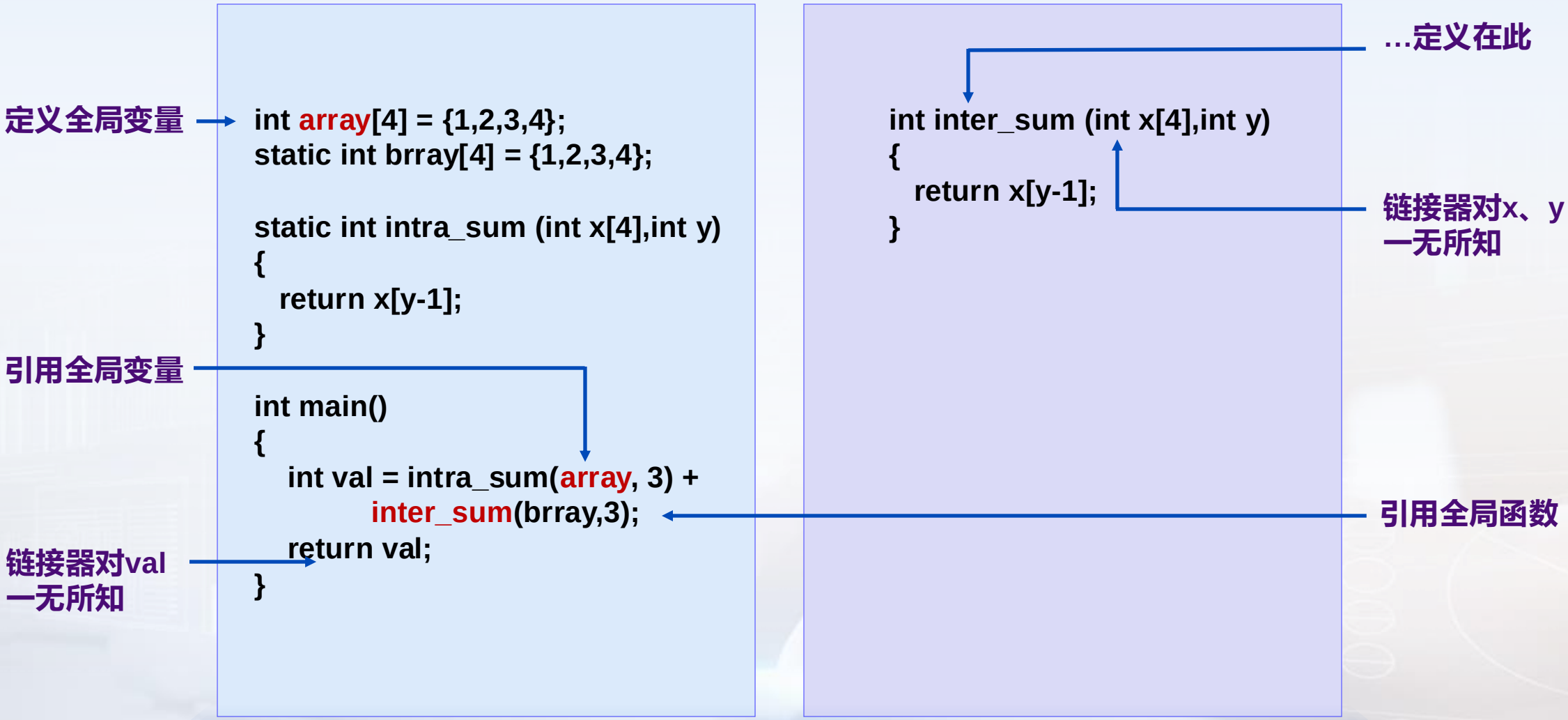
外部符号

- 某个模块引用的由其他模块定义的全局符号

局部符号

- 由某个模块定义且仅有该模块引用的符号
- 比如静态全局变量
  - 这与程序的局部变量不是一个概念

符号解析


定义全局变量
int array[4] = {1,2,3,4};
static int brray[4] = {1,2,3,4};
static int intra_sum (int x[4],int y)
{
return x[y-1];
}
引用全局变量
int main()
{
int val = intra_sum(array, 3) +
inter_sum(brray,3);
return val;
}
链接器对val
一无所知
…定义在此
int inter_sum (int x[4],int y)
{
return x[y-1];
}
链接器对x、y
一无所知
引用全局函数

局部符号

（C语言）局部非静态变量vs.局部静态变量

- 局部非静态变量:存储在堆栈或某些寄存器中
- 局部静态变量:存储在.bss或.data段中

```
int f()
{
    static int x = 2;
    return x;
}

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

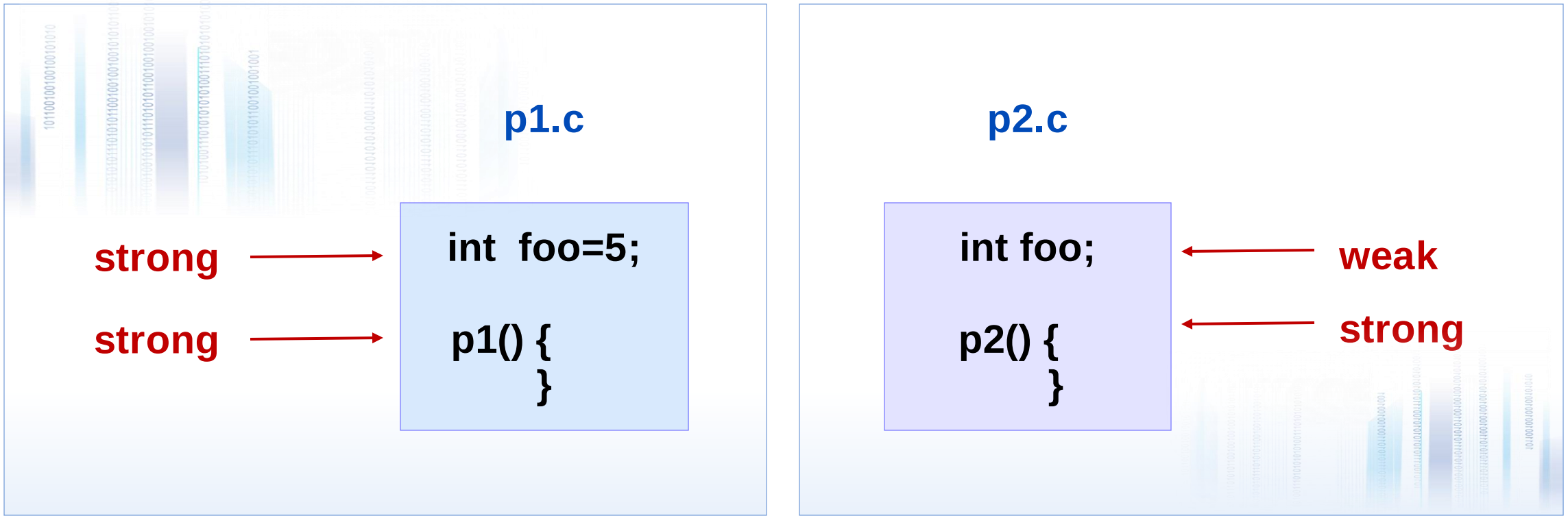
编译器在.data中为x的每个定义分配空间

在符号表中创建具有唯一名称的局部符号，例如x.1和x.2。

链接器如何解决重复的符号定义问题

程序符号有强(strong)弱(weak)之分

- **Strong**: 过程以及初始化的全局变量
- **Weak**: 未初始化的全局变量



解决规则

规则1：不允许使用多个同名强符号

- 每项只能定义一次
- 否则：链接错误

规则2：给定一个强符号和多个弱符号，则选择强符号

- 对弱符号的引用被解析为强符号

规则3：如果有多个弱符号，选择任意一个

- 可以通过gcc的编译开关 –fno-common来关闭此属性（即给出警告）

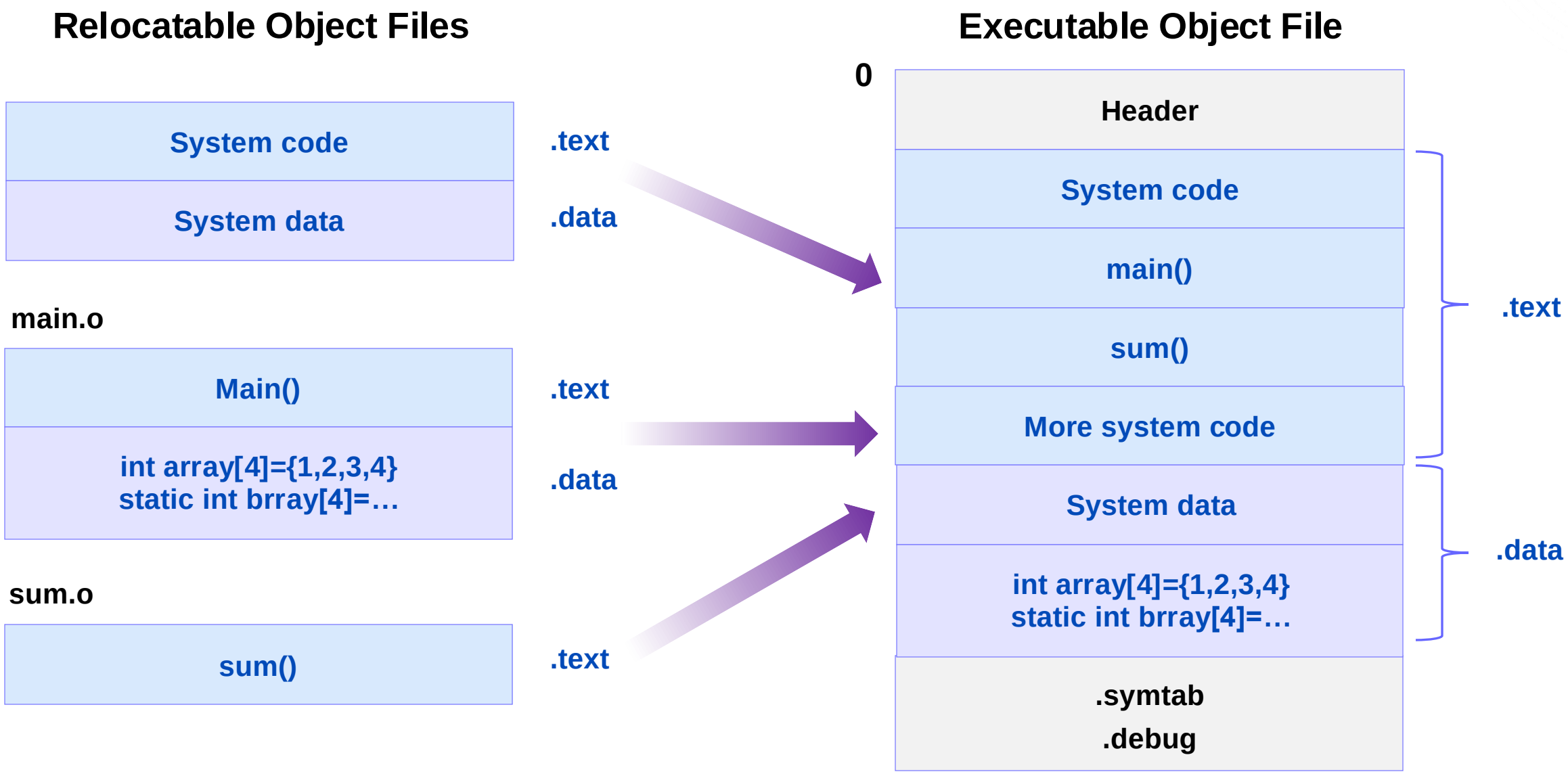
全局变量使用建议

尽量避免使用全局变量

如果必须使用

- 尽量使用static
- 定义时要初始化
- 如果引用外部全局变量，请使用extern

代码与数据重定位


Relocatable Object Files
System code
System data
main.o
Main()
int array[4]={1,2,3,4}
static int brray[4]=…
sum.o
sum()
.text
.data
.text
.data
.text
0
Executable Object File
Header
System code
main()
sum()
More system code
System data
int array[4]={1,2,3,4}
static int brray[4]=…
.symtab
.debug
.text
.data

重定位信息 (main.o　sum.o)

```
.....
0000000000000008 <main>:
  8:        53                    push   %rbx
  9:        be 03 00 00 00        mov    $0x3,%esi
  e:        bf 00 00 00 00        mov    $0x0,%edi
                                  f: R_X86_64_32       array  #relocation entry
 13:        e8 e8 ff ff ff        callq  0 <intra_sum>
 18:        89 c3                 mov    %eax,%ebx
 1a:        be 03 00 00 00        mov    $0x3,%esi
 1f:        bf 00 00 00 00        mov    $0x0,%edi
                                  20: R_X86_64_32      brray  #relocation entry
 24:        b8 00 00 00 00        mov    $0x0,%eax
 29:        e8 00 00 00 00        callq  2e <main+0x26>
                                  2a: R_X86_64_PC32    inter_sum - 0x4 # ...
 2e:        01 d8                 add    %ebx,%eax
 30:        5b                    pop    %rbx
 31:        c3                    retq
```

Why “-4”?

main.o

```
int array[4] = {1,2,3,4};
static int brray[4] = {1,2,3,4};

static int intra_sum (int x[4],int y)
{
   return x[y-1];
}

int main()
{
    int val = intra_sum(array, 3) +
              inter_sum(brray,3);
    return val;
}
```

objdump –r –D

```
0000000000000000 <inter_sum>:
  0:        48 63 f6              movslq %esi,%rsi
  3:        8b 44 b7 fc           mov    -0x4(%rdi,%rsi,4),%eax
  7:        c3                    retq
```

sum.o

重定位信息 (prog)

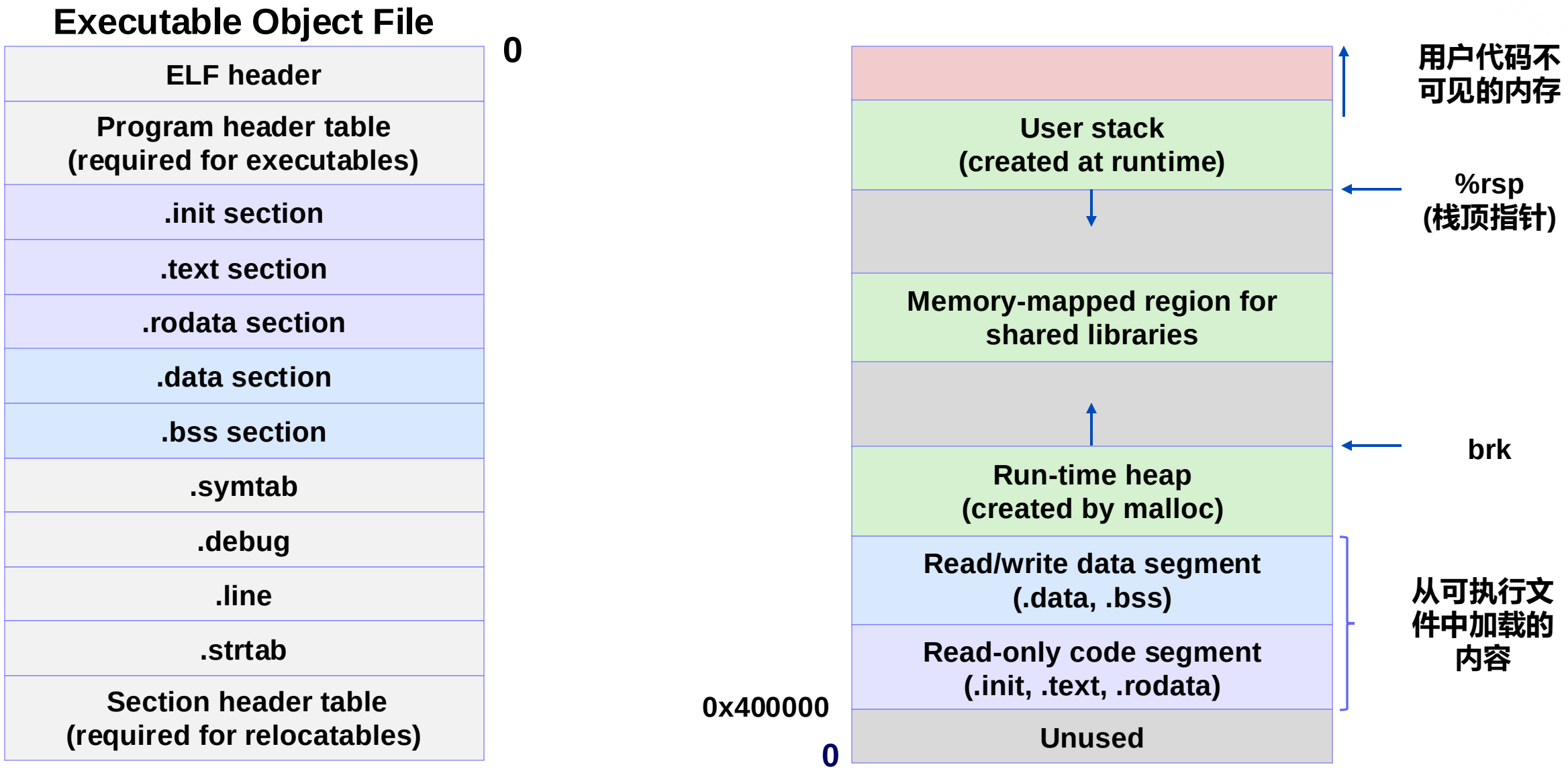
```
00000000004004d6 <inter_sum>:
  4004d6: 48 63 f6          movslq %esi,%rsi
  4004d9: 8b 44 b7 fc       mov  -0x4(%rdi,%rsi,4),%eax
  4004dd: c3                retq
  ......
00000000004004e6 <main>:
  4004e6: 53                push   %rbx
  4004e7: be 03 00 00 00    mov    $0x3,%esi
  4004ec: bf 40 10 60 00    mov    $0x601040,%edi
  4004f1:  e8 e8 ff ff ff   callq  4004de <intra_sum>
  4004f6:  89 c3            mov    %eax,%ebx
  4004f8:  be 03 00 00 00   mov    $0x3,%esi
  4004fd:  bf 30 10 60 00   mov    $0x601030,%edi
  400502: b8 00 00 00 00    mov    $0x0,%eax
  400507: e8 ca ff ff ff    callq  4004d6 <inter_sum>
  40050c: 01 d8             add    %ebx,%eax
  40050e: 5b                pop    %rbx
  40050f:  c3               retq
```

objdump –r -D prog

```
0000000000601030<brray>:
601030:          01 00
601032:          00 00
601034:          02 00
601036:          00 00
601038:          03 00
60103a:          00 00
60103c:          04 00
                 …
0000000000601040<array>:
601040:          01 00
601042:          00 00
601044:          02 00
601046:          00 00
601048:          03 00
60104a:          00 00
60104c:          04 00
```

针对 inter_sum() 使用PC相对寻址（PC-relative addressing）： 0x4004d6 = 0x40050c + 0xffffffca

运行前装载Executable Object Files



数据、函数、指令“定位”小结

	编译	链接（静态）	说明
局部变量(非静态)	√		存于栈内 or 寄存器 or 被优化掉
全局变量		√	各个模块(.o文件)的数据段、代码段合并后，方能确定绝对地址
函数参数	√		通过指定的寄存器与栈传递
函数返回值	√		%rdx + %rax
跳转指令（直接）	√		相对于PC的offset
函数返回地址	√		存于栈顶
静态函数入口地址	√		相对于PC的offset
全局函数入口地址		√	各个模块(.o文件)的数据段、代码段合并后，方能确定绝对地址

非静态全局函数/变量是可以被多个模块引用的，而静态的不会，这是实现“定位”时需要考虑的

将常用的函数打包

如何打包?

数学运算，I/O操作，内存管理，字符串操作等等常见功能

Option 1

- 程序员将这一“大”对象文件链入自己的程序
- 空间/时间效率不高

Option 2

- 程序员有选择性的链接所需的对象文件
- 高效；但是程序员的负担较重

解决方案: 静态库文件

静态库文件(.a 文件)

01

将多个相关的重定位对象文件集成为一个单一的带索引的文件 (称为归档文件，archive file)

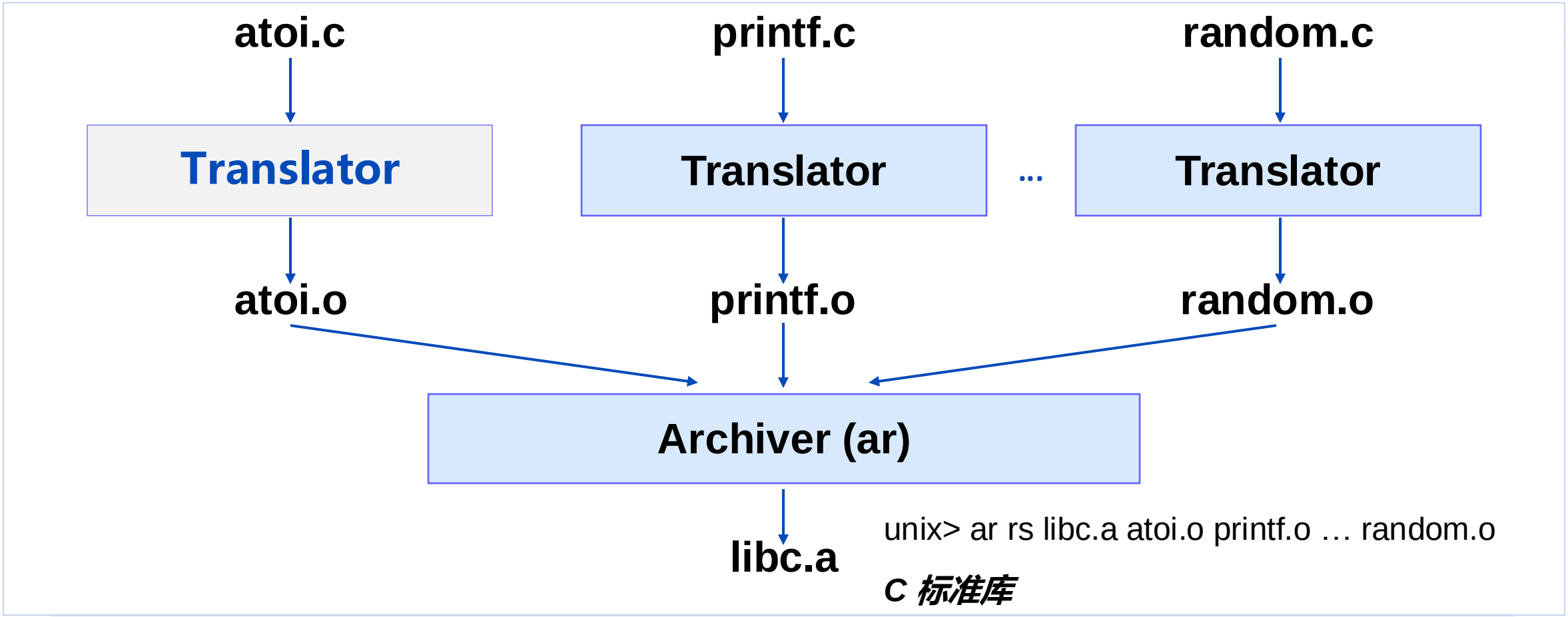
02

增强链接器的功能使之能够在归档文件中解析外部符号

03

如果归档文件中的某个成员解析了外部符号，就将其链接入执行文件

创建静态库文件



- 归档文件可以以增量方式更新
- 重编译更新过的源文件,并替换归档文件中的对应部分

通常使用的库文件（举例）

libc.a (C标准库)

- 8 MB 大小，包含 1392 个目标文件
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

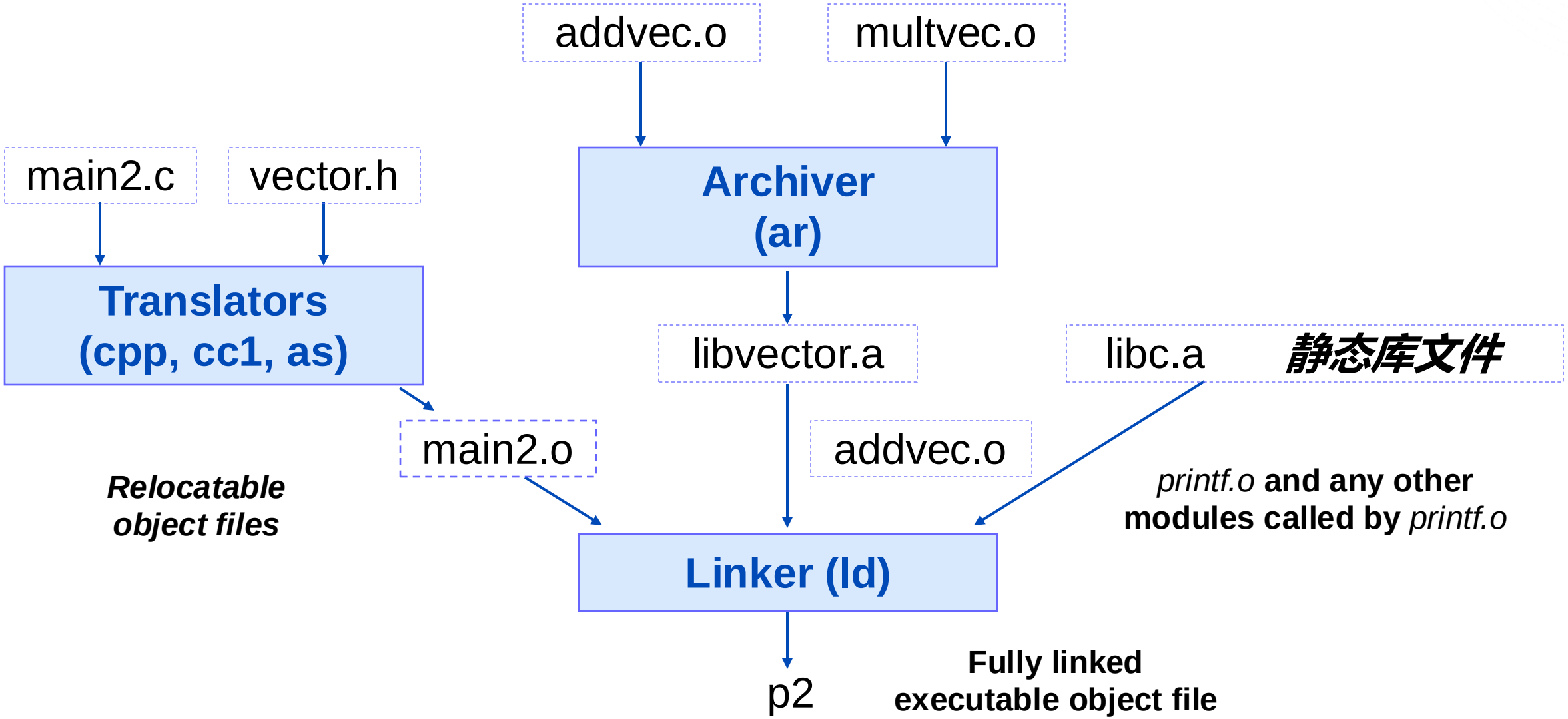
```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

libm.a (C数学运算库)

- 1 MB 大小，包含 401 个目标文件
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% 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
…
```

与静态库（static library）链接



共享库文件

静态库文件有其劣势

- 不同的执行文件中会重复包含有所需的库文件函数或者数据
- 运行时内存中也会有重复部分
- 库文件的细微变动需要所有相关执行文件进行重链接

更好的方案：共享库文件方式

- 特殊类型的重定向对象文件，可以被装载入内存后进行动态链接；链接可以在装载时或者运行时完成
- Windows系统下被称为DLL文件

补充：X86-32下共享库中的全局变量寻址

```
int a;
int b;

void bar()
{
   a = 1;
   b = 2;
}
```

```
00000510 <bar>:
 510:   55                      push   %ebp
 511:   89 e5                   mov    %esp,%ebp
 513:   e8 20 00 00 00          call   538 <__x86.get_pc_thunk.ax>
 518:   05 e8 1a 00 00          add    $0x1ae8,%eax    # 0x2000, global offset table
 51d:   8b 90 f4 ff ff ff       mov    -0xc(%eax),%edx    # index in the GOT
 523:   c7 02 01 00 00 00       movl   $0x1,(%edx)          # a
 529:   8b 80 e8 ff ff ff       mov    -0x18(%eax),%eax
 52f:   c7 00 02 00 00 00       movl   $0x2,(%eax)          # b
 535:   90                      nop
 536:   5d                      pop    %ebp
 537:   c3                      ret

00000538 <__x86.get_pc_thunk.ax>:
 538:   8b 04 24                mov    (%esp),%eax
 53b:   c3                      ret
```

```
Disassembly of section .got:

00001fe4 <.got>:
        ...

Disassembly of section .got.plt:

00002000 <_GLOBAL_OFFSET_TABLE_>:
```

```
gcc -fPIC  -shared  -m32  -Og  -fno-omit-frame-pointer pic.c  -o libpic.so
    //Position Independent Code（地址无关代码）
```

补充：X86-32下共享库中的全局变量寻址

事实

- （共享库）代码段中的任意指令与数据段中的任意变量之间的 ”距离”在运行时是一个常量，与代码和数据加载的**绝对内存位置无关**

方法（编译器）

- 为了利用这一特点，编译器在数据段的开头创建了一个全局偏移表（GOT），每个全局数据对象(全局变量)都对应一个偏移表项
- 编译器同时为GOT中的每个表项生成了一个重定位记录
- 每个包含全局数据引用的目标模块都有其自己的GOT

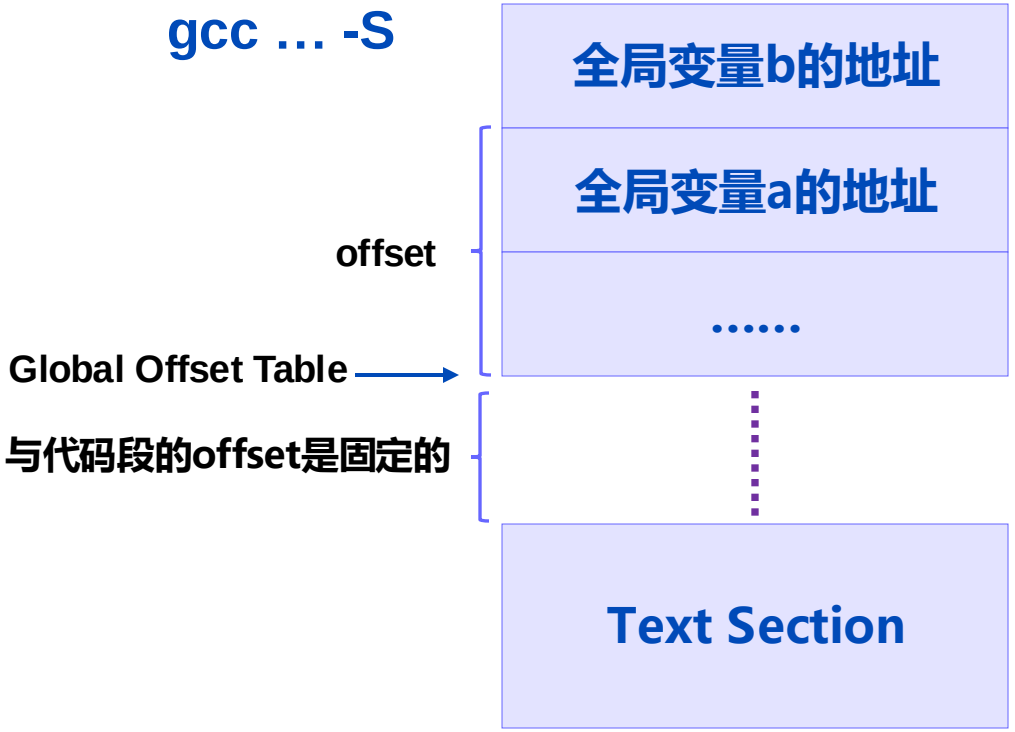
方法（链接器）

- 动态链接器**重定位GOT中的每个表项，使其包含正确的绝对地址**

在运行时，全局变量通过GOT被间接引用

```
bar:
        pushl   %ebp
        movl    %esp, %ebp
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        movl    a@GOT(%eax), %edx
        movl    $1, (%edx)
        movl    b@GOT(%eax), %eax
        movl    $2, (%eax)
        nop
        popl    %ebp
        ret
```

gcc … -S



补充：X86-64下的一种方案

```
00000000000006c0 <bar>:
 6c0:   55                      push   %rbp
 6c1:   48 89 e5                mov    %rsp,%rbp
 6c4:   48 8b 05 15 09 20 00    mov    0x200915(%rip),%rax        # 200fe0
 6cb:   c7 00 01 00 00 00       movl   $0x1,(%rax)
 6d1:   48 8b 05 f8 08 20 00    mov    0x2008f8(%rip),%rax        # 200fd0
 6d8:   c7 00 02 00 00 00       movl   $0x2,(%rax)
 6de:   90                      nop
 6df:   5d                      pop    %rbp
 6e0:   c3                      retq
```

```
bar:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    a@GOTPCREL(%rip), %rax
        movl    $1, (%rax)
        movq    b@GOTPCREL(%rip), %rax
        movl    $2, (%rax)
        nop
        popq    %rbp
        ret
```

```
Contents of section .got:
 200fc8 00000000 00000000 00000000 00000000  ................
 200fd8 00000000 00000000 00000000 00000000  ................
 200fe8 00000000 00000000 00000000 00000000  ................
 200ff8 00000000 00000000                    ........
```

static与非static的全局变量的定位方式有何不同?

```
static  int a,b;
int c,d;

void bar()
{
   a = 1;
   b = 2;
   c = 3;
   d = 4;
}
```

```
bar:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $1, a(%rip)
        movl    $2, b(%rip)
        movq    c@GOTPCREL(%rip), %rax
        movl    $3, (%rax)
        movq    d@GOTPCREL(%rip), %rax
        movl    $4, (%rax)
        nop
        popq    %rbp
        ret
```