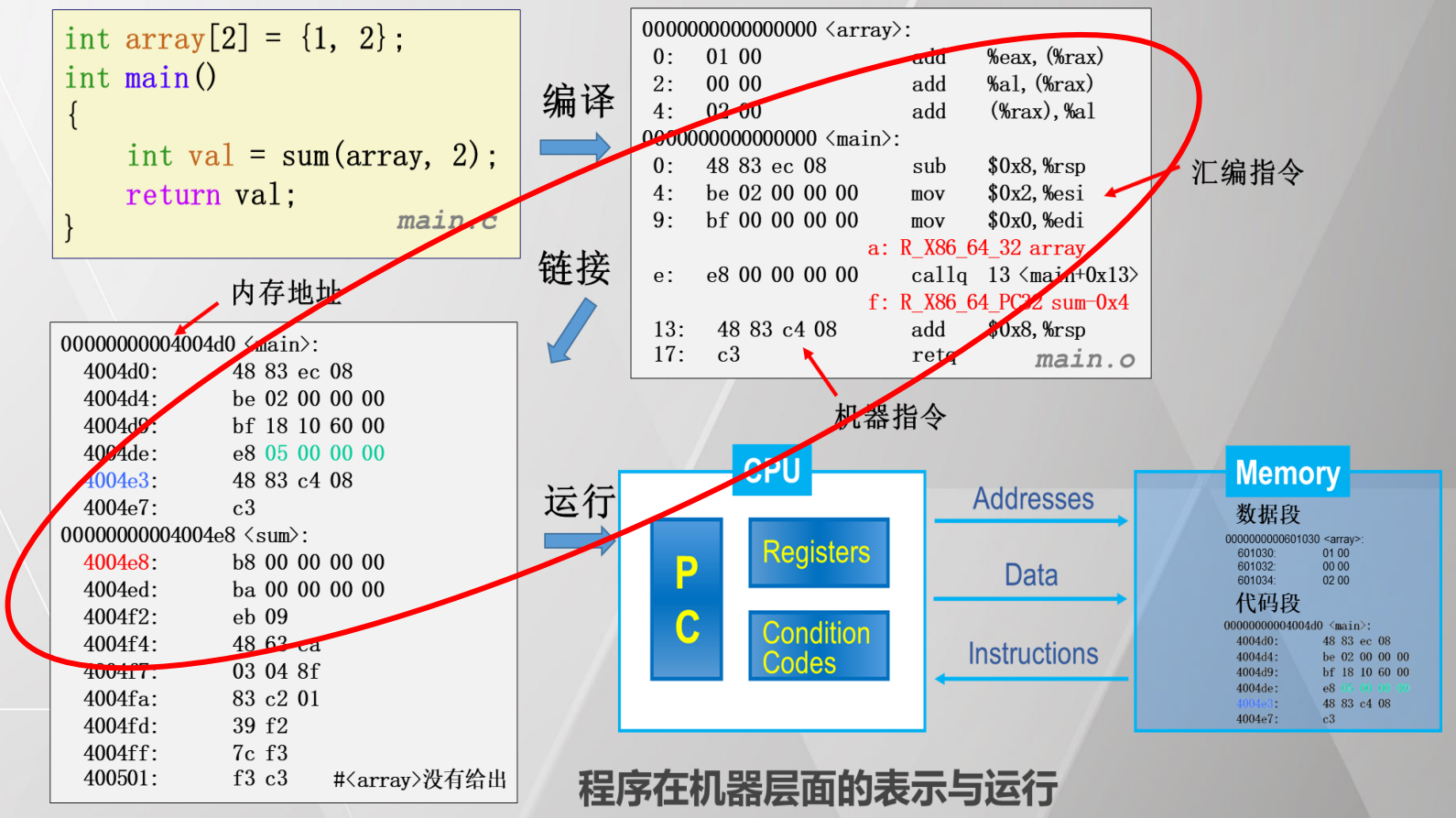
# 汇编语言程序设计

## 程序链接（linking）

# C程序在硬件层面的表示

- 数据/代码的内存地址定位
  - 链接（第九讲）
- 数据/代码的内存布局
  - 栈、堆等各类数据段以及代码段的layout（第十讲）
  - 缓冲区溢出等（第十讲）
- 讲解基本调试工具（GDB）的使用




程序在机器层面的表示与运行

# C语言示例程序

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

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

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

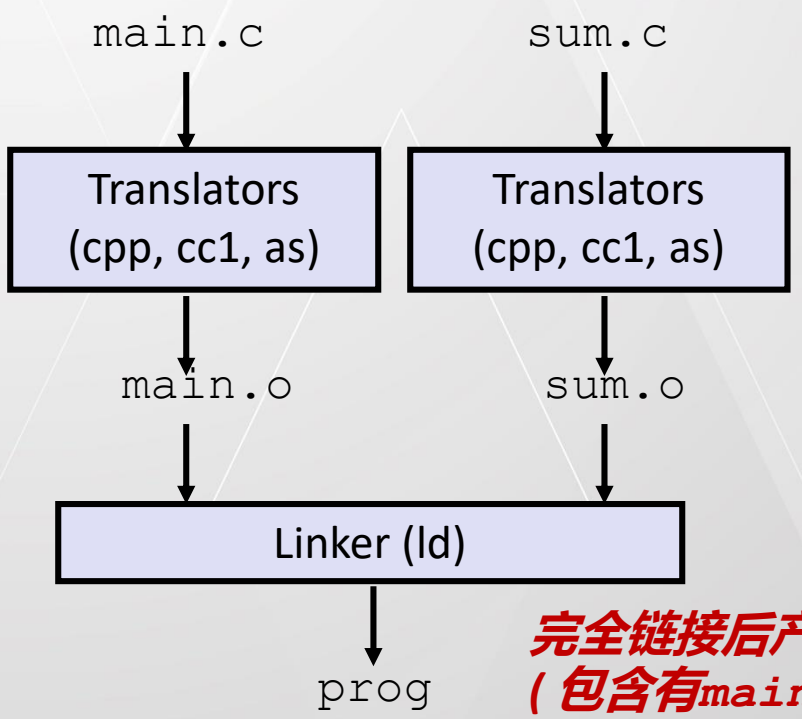
***main.c***

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

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

***sum.c***

# 静态链接(Static Linking)

- 编译与链接
  - *unix> gcc -Og -o prog main.c sum.c*
  - *unix> ./prog*




***源文件***

***编译生成的可重定位对象（Relocatable object）文件***

***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**
  - e.g., 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() {…}  /* define symbol swap */
  - swap();  /* reference symbol swap */
  - int *xp = &x;  /* define symbol xp, reference x */

- **编译器将符号定义存储在符号表（*symbol table*）中.**
  - Symbol table is an array of structs
  - Each entry includes name, size, and location of symbol.

- **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

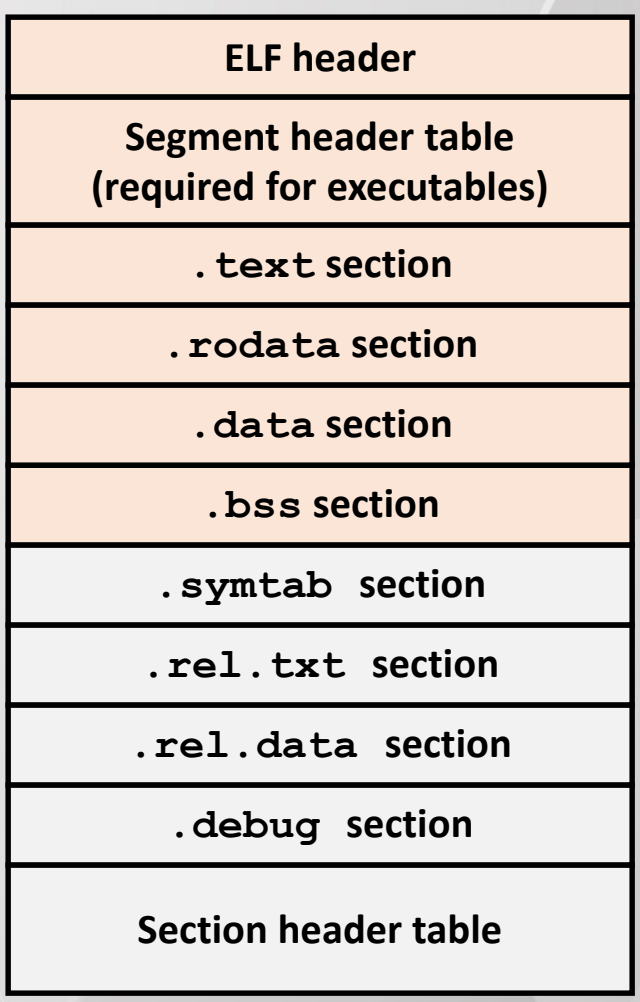
- 重定向对象文件 (.o 文件)
  - 含有一定格式的代码与数据内容，可以与其它重定向对象文件一起集成为执行文件
    - 一个 .o 文件由唯一的一个源文件生成
- 执行文件(a.out 文件)
  - 含有一定格式的代码与数据内容，可以直接被装载入内存并执行
- 共享对象文件 (.so 文件)
  - 特殊类型的重定向对象文件，可以被装载入内存后进行动态链接；链接可以在装载时或者运行时完成
  - Windows系统下被称为DLL文件

# 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

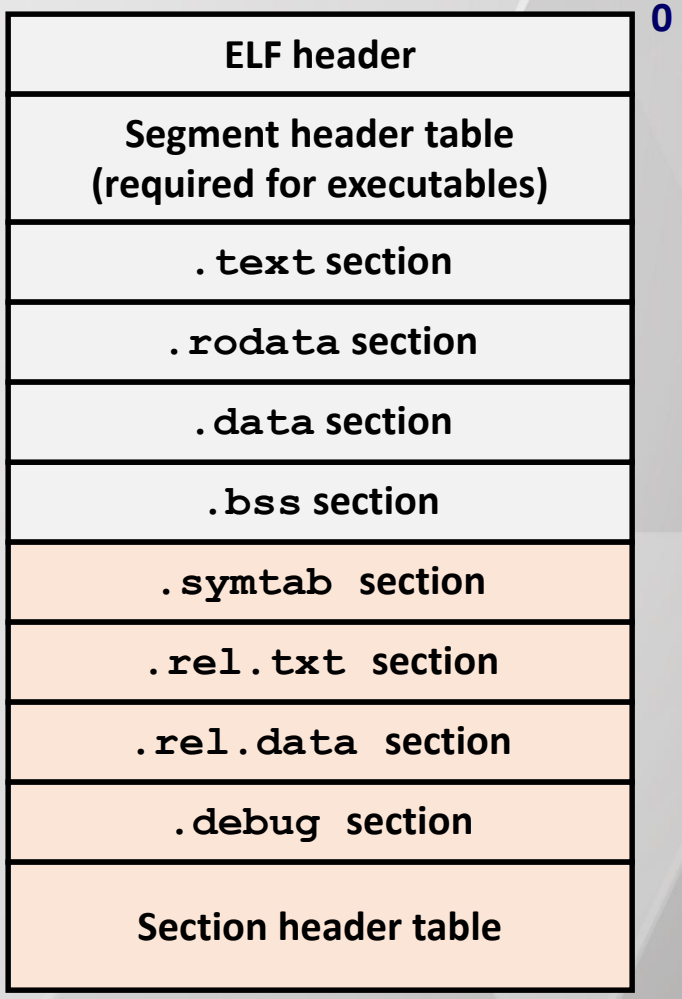
- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **.text section**
  - Code
- **.rodata section**
  - Read only data: jump tables, ...
- **.data section**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




*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)

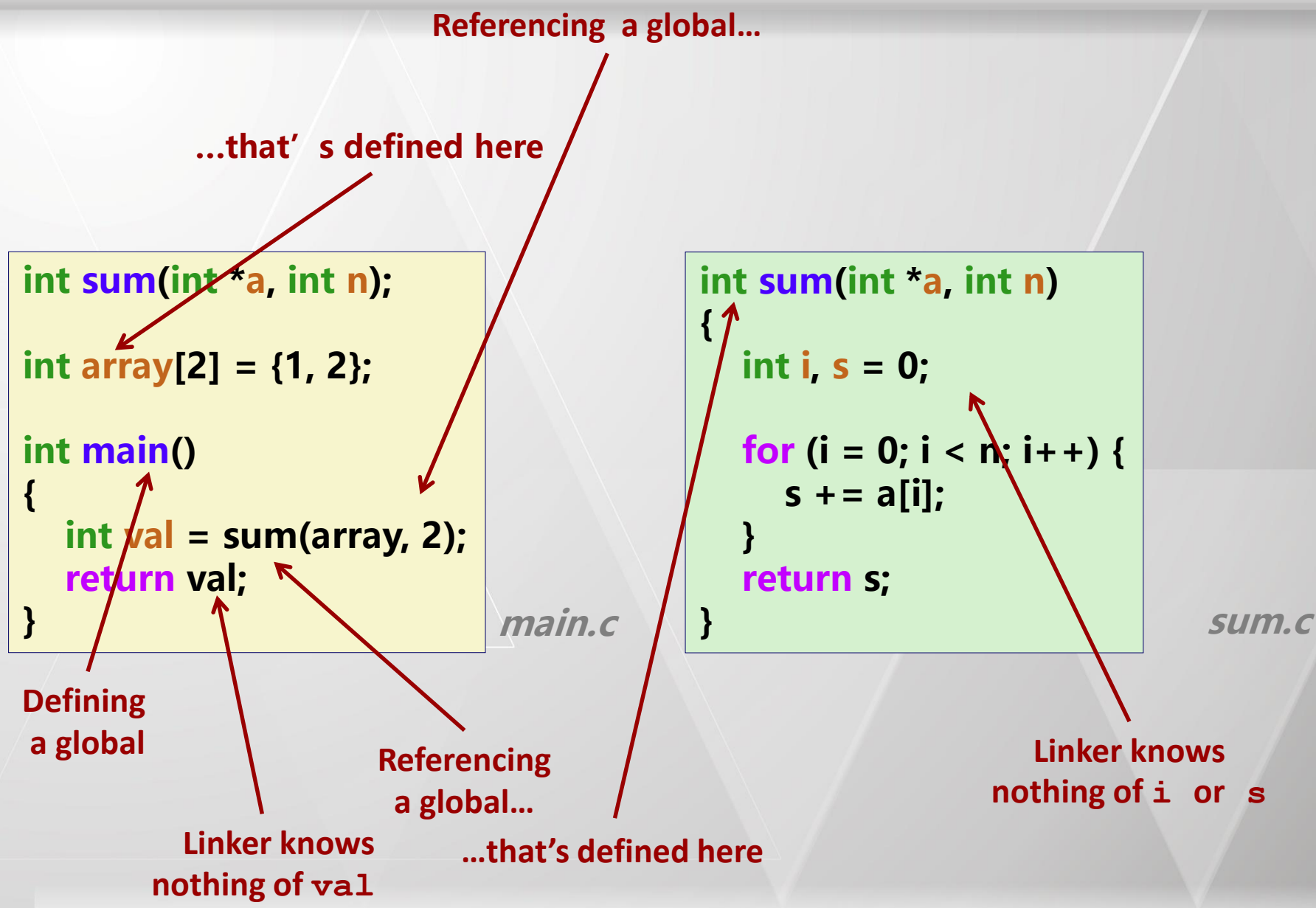
- **.symtab section**
  - Symbol table
  - Global procedure and variable names
  - Section names and locations
- **.rel.text section**
  - Relocation info for **.text** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **.rel.data 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



# 链接符号

- **全局符号**
  - 某一个模块定义的、且可以被其它模块引用的变量或者函数符号.
  - 比如non-**static** C 函数以及non-**static** 全局变量.
- **外部符号**
  - 某个模块引用的由其它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**static** C 函数以及**static** 全局变量.
    - **这与程序的局部变量不是一个概念**

# 符号解析


Referencing a global…
…that’ s defined here
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
Defining a global
Linker knows nothing of val
Referencing a global…
…that’s defined here
Linker knows nothing of i or s

# 局部符号

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

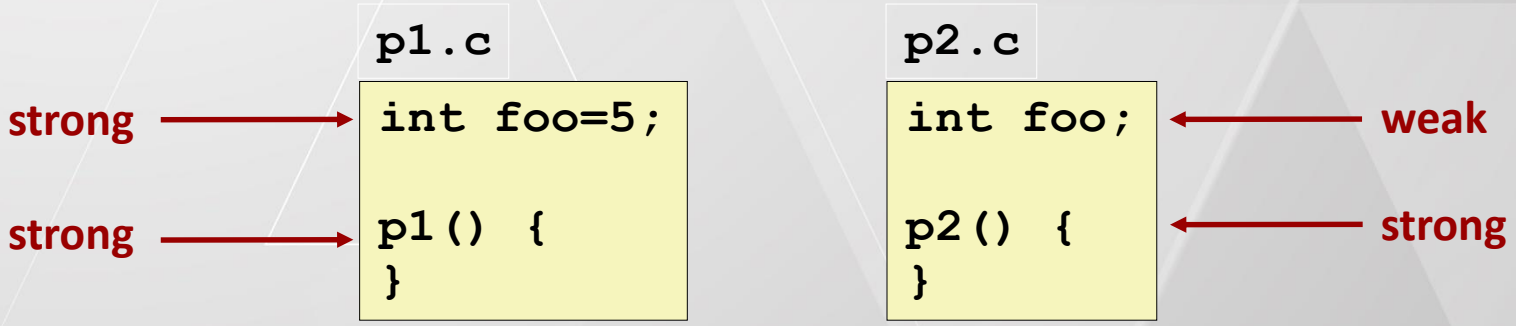
```
int f()
{
    static int x = 2;
    return x;
}

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

- **Program symbols are either *strong* or *weak***
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals



# Linker's Symbol Rules

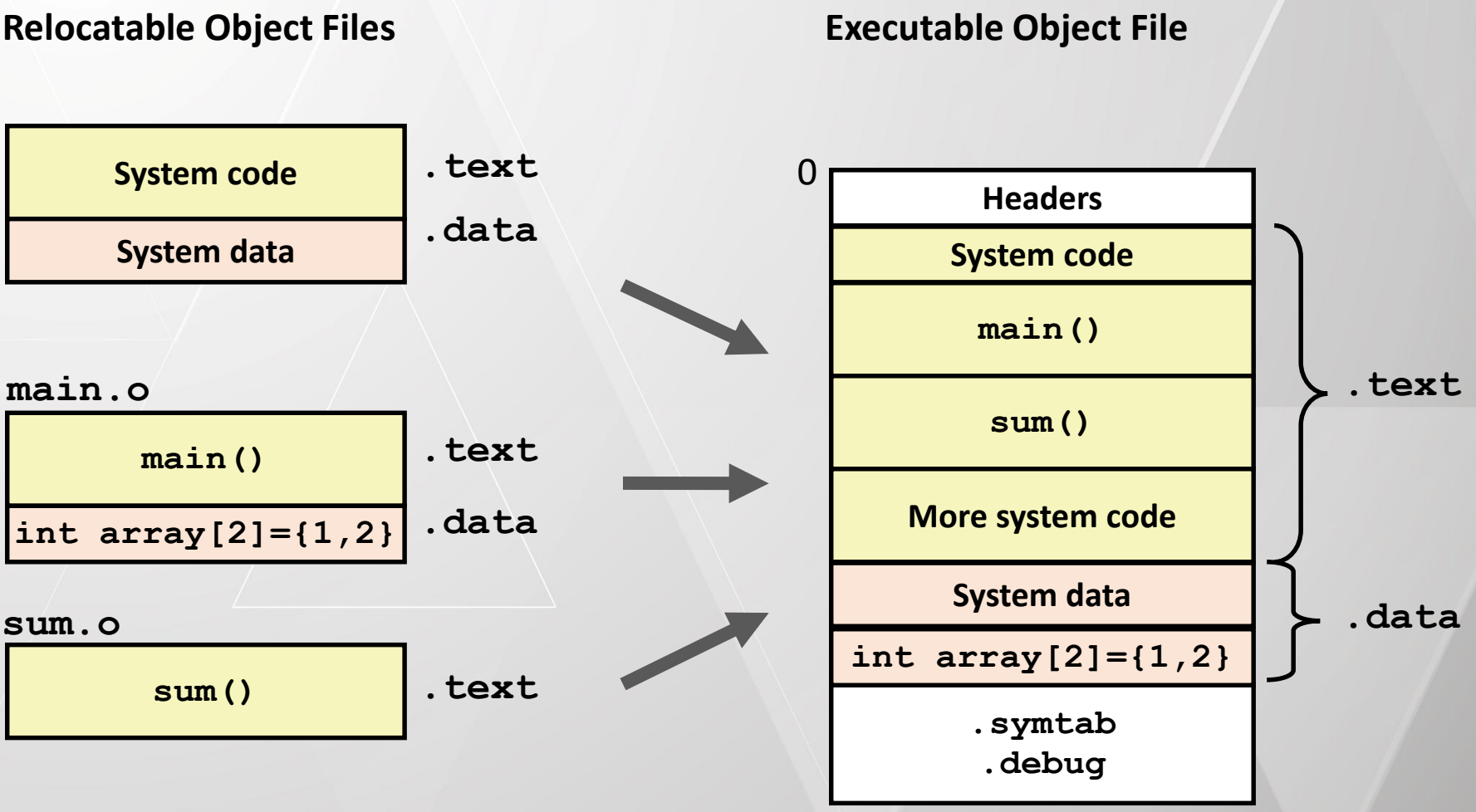
- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple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

# Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable

# 代码与数据重定位


Relocatable Object Files
System code
System data
.text
.data
main.o
main()
int array[2]={1,2}
.text
.data
sum.o
sum()
.text
Executable Object File
0
Headers
System code
main()
sum()
More system code
System data
int array[2]={1,2}
.symtab
.debug
.text
.data

# 重定位信息 (main)

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

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

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi       # %edi = &array
                        a: R_X86_64_32 array          # Relocation entry

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

Why "-4"?

Source: objdump -r -d main.o

# 重定位信息

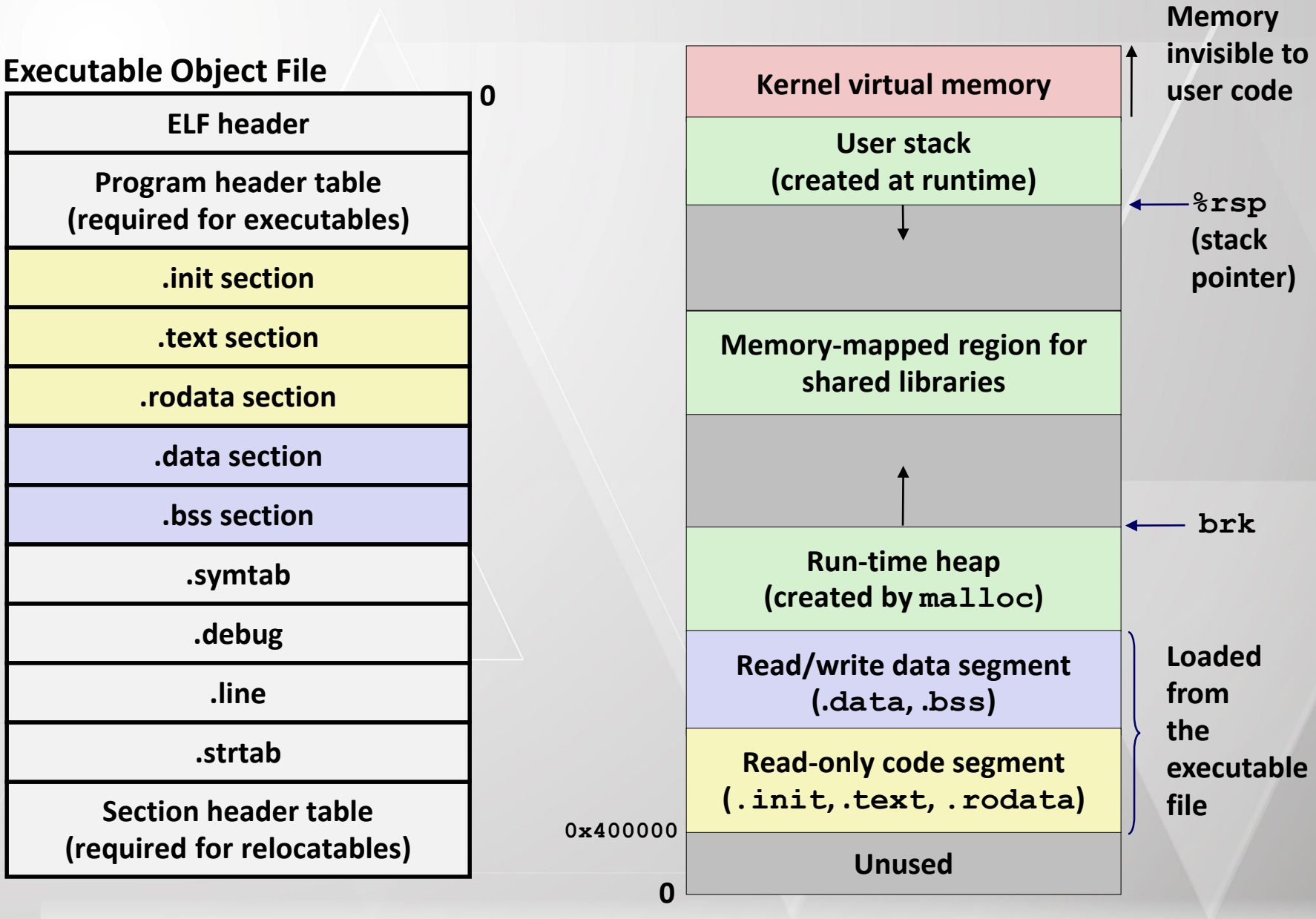
```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08          sub    $0x8,%rsp
  4004d4:       be 02 00 00 00       mov    $0x2,%esi
  4004d9:       bf 18 10 60 00       mov    $0x601018,%edi   # %edi = &array
  4004de:       e8 05 00 00 00       callq  4004e8 <sum>     # sum()
  4004e3:       48 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
```

```
0000000000601018 <array>:
  601018:       01 00                add    %eax,(%rax)
  60101a:       00 00                add    %al,(%rax)
  60101c:       02 00                add    (%rax),%al
     ...
```

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

Source: objdump -dx prog

# Loading Executable Object Files



# 将常用的函数打包

- **如何打包?**
  - Math, I/O, memory management, string manipulation, etc.

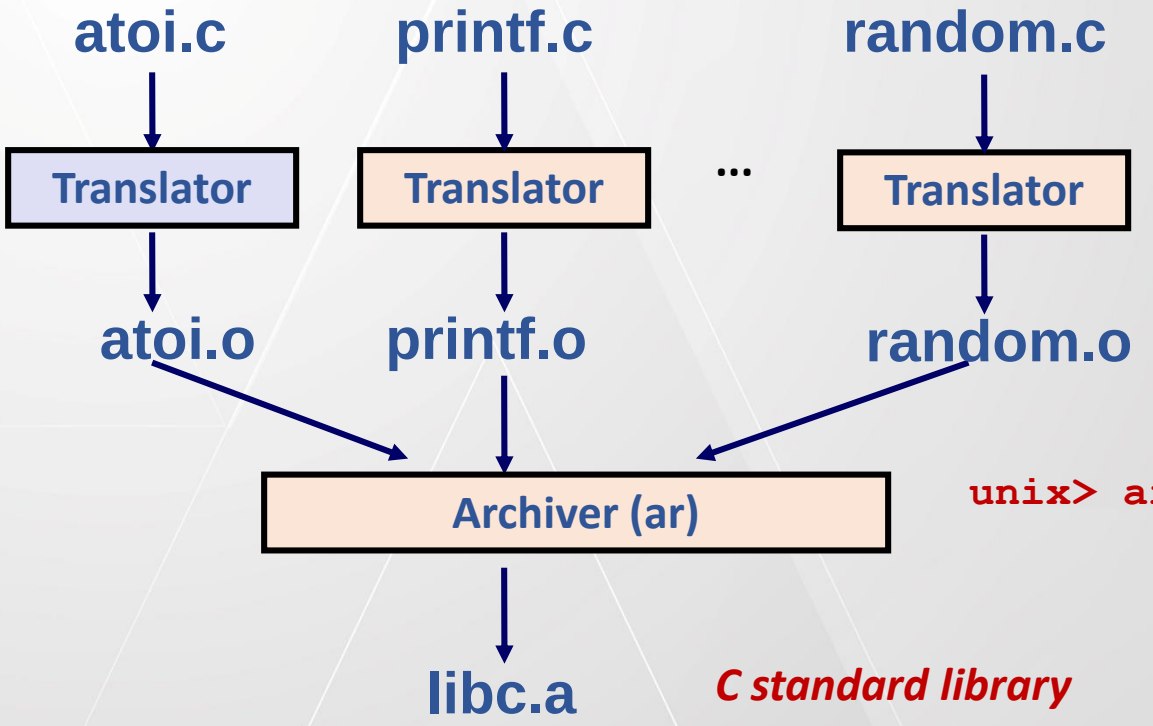
  - **Option 1:** 将所有的函数都放入同一个源文件
    - 程序员将这一“大”对象文件链入自己的程序
    - 空间/时间效率不高

  - **Option 2:** 每个函数一个文件
    - 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)**
  - 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).
  - 增强链接器的功能使之能够在归档文件中解析外部符号.
  - 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




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

- 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

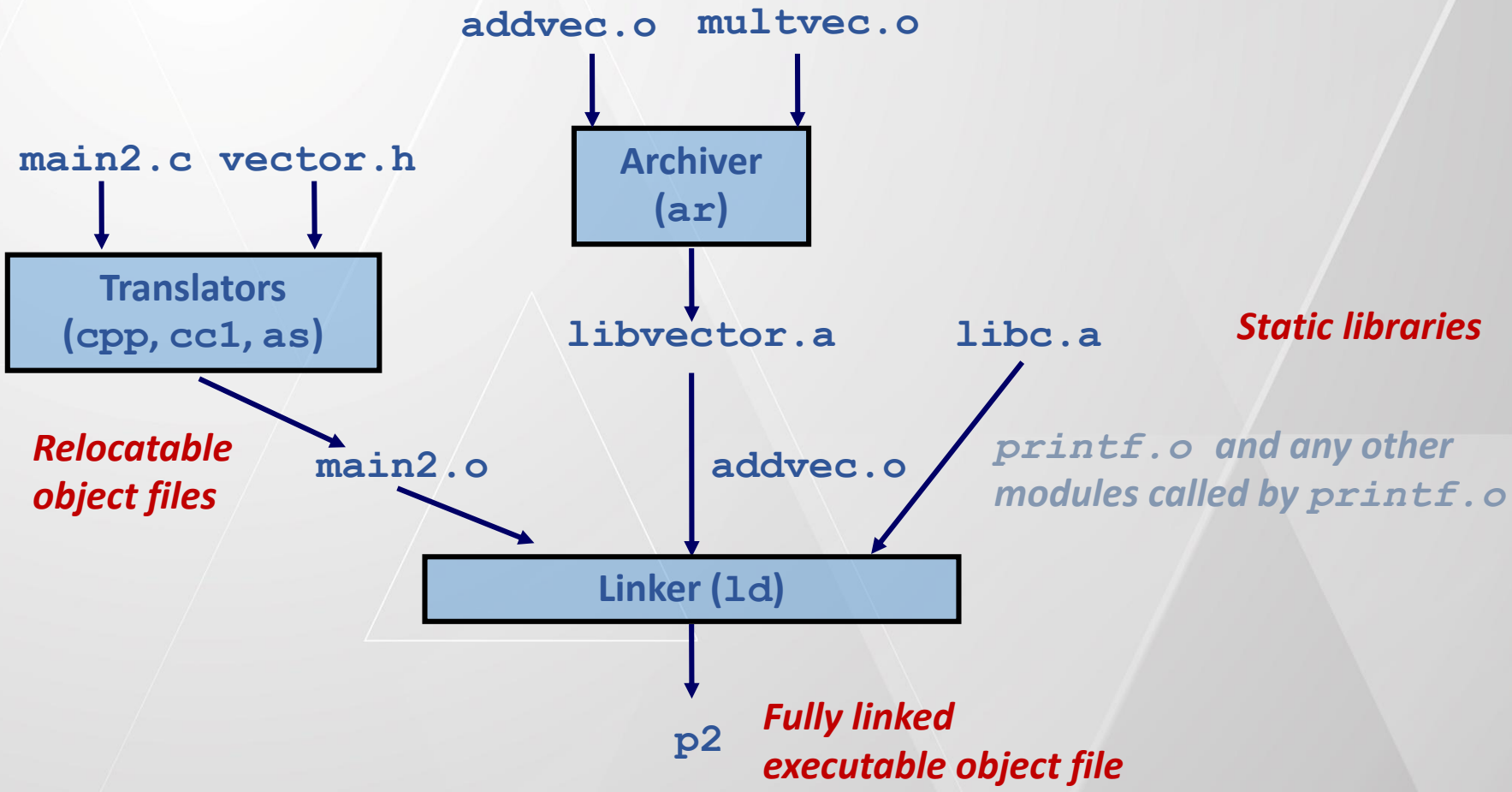
# 通常使用的库文件（举例）

- **libc.a (the C standard library)**
  - 8 MB archive of 1392 object files.
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- **libm.a (the C math library)**
  - 1 MB archive of 401 object files.
  - floating point math (sin, cos, tan, log, exp, sqrt, ...)

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

```
% ar -t /usr/lib/libm.a | sort
...
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
...
```

# 与静态库链接


addvec.o
multvec.o
main2.c
vector.h
Archiver
(ar)
Translators
(cpp, cc1, as)
libvector.a
libc.a
Static libraries
Relocatable
object files
main2.o
addvec.o
printf.o and any other
modules called by printf.o
Linker (ld)
p2
Fully linked
executable object file

# 共享库文件

- **静态库文件有其劣势:**
  - 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

# X86-32下的全局变量寻址

```
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 pic.c -o libpic.so**

// Position Independent Code

## 事实

- 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都是一个常量，而与代码和数据加载的绝对内存位置无关

## 方法（编译器）

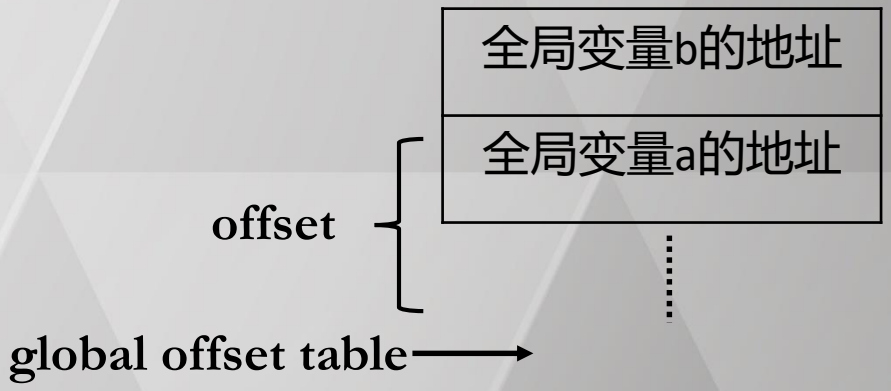
- 为了利用这一特点，编译器在数据段的开头创建了一个全局偏移表（GOT），目标模块所引用的每个全局数据对象都对应一个表项
- 编译器同时为GOT中的每个表项生成了一个重定位记录
- 每个包含全局数据引用的目标模块都有其自己的GOT

## 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，每个全局变量通过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
```



# X86-64下的一种方案

```
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
```

```
0000000000006c0 <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
```

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