

# 程序的结构、装载与执行机理

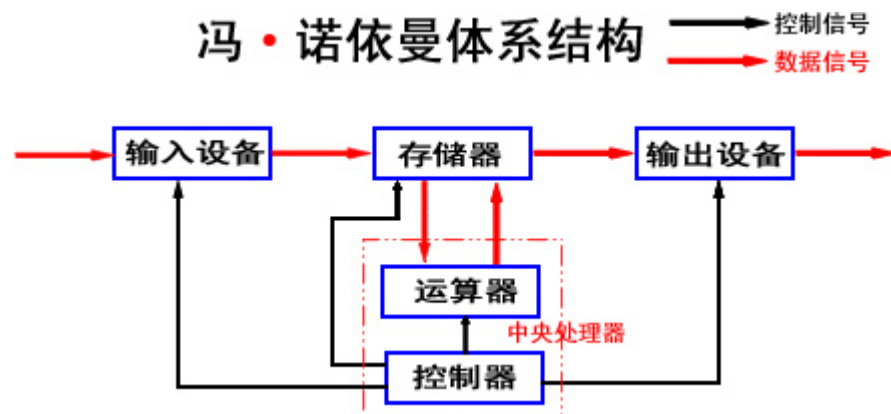
# 可执行文件的结构链接及运行

## ■ 主要内容

- 程序的编译
- 目标文件格式
- 链接和静态链接概念
- 符号及符号表、符号解析
- 使用静态库链接
- 重定位信息及重定位过程
- 可执行文件的存储器映像
- 可执行文件的加载
- 共享（动态）库链接

# 冯诺依曼体系结构

- 冯·诺依曼的系统结构就是对图灵机抽象描述的一种可行而且有效的设计
- 核心理念：计算机应该是由程序来控制的
  - 程序及数据的存储
  - 计算状态



# 程序的执行

可执行文件  
(保存于文件系统)

代码

数据

内存映射

内存镜像

栈

堆

取指、译指、执行

CPU

数据的读和写

- 冯氏结构核心理念
  - 计算机是由程序来控制的

## 2.1 程序的生成

```
void main()  
{  
    printf (" hello world!\n");  
}
```

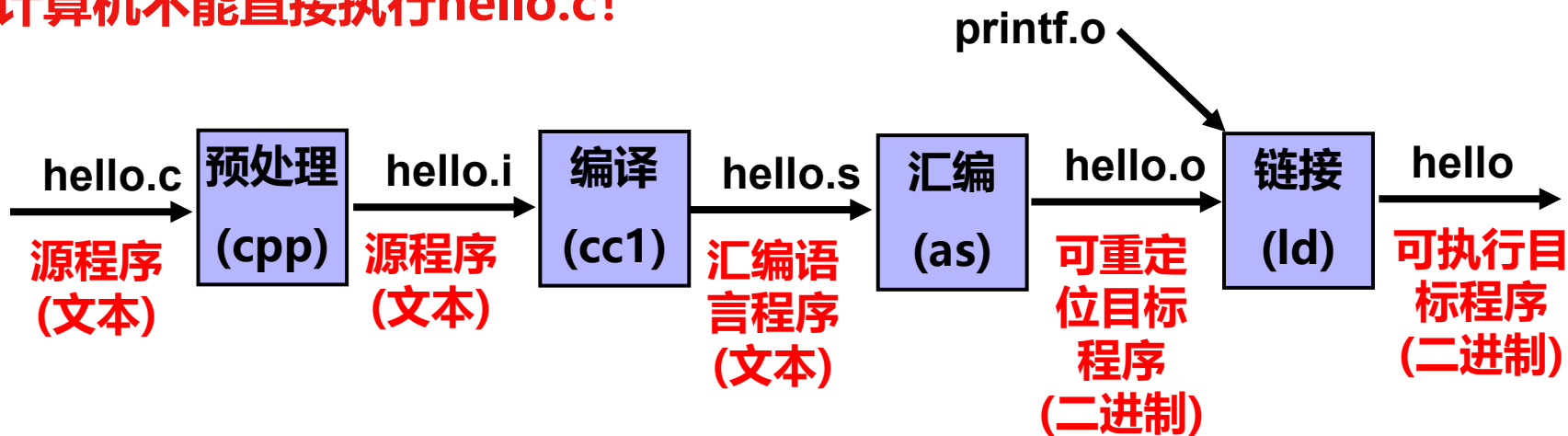


- 源代码编写的程序是如何转换为可执行代码的?
- 代码执行的机器模型又是什么?

# 一个典型程序的转换处理过程

```
void main()
{
    printf (" hello world!\n");
}
```

计算机不能直接执行hello.c!



## 2.1 程序的编译

源文件  
hello.c

编译

目标文件  
hello.o

预处理：  
符号、宏  
定义等

编译：词法分  
析、语法分析、  
规则检查、翻  
译成目标代码

汇编：将汇  
编语言代码  
翻译成目标  
机器指令

链接：将  
多个目标  
文件、共  
享库等链  
接在一起

```
void main()  
{  
    printf (" hello world!\n");  
}
```

```
0101100101010  
101010101010  
001101010110  
010111101010
```

## 2.1 程序的编译

### Dump of assembler code for function main:

```
0x08048394 <main+0>: lea 0x4(%esp),%ecx
0x08048398 <main+4>: and $0xffffffff0,%esp
0x0804839b <main+7>: pushl 0xffffffffc(%ecx)
0x0804839e <main+10>: push %ebp
0x0804839f <main+11>: mov %esp,%ebp
0x080483a1 <main+13>: push %ecx
0x080483a2 <main+14>: sub $0x54,%esp
0x080483a5 <main+17>: mov %gs:0x14,%eax
0x080483ab <main+23>: mov %eax,0xffffffff8(%ebp)
```

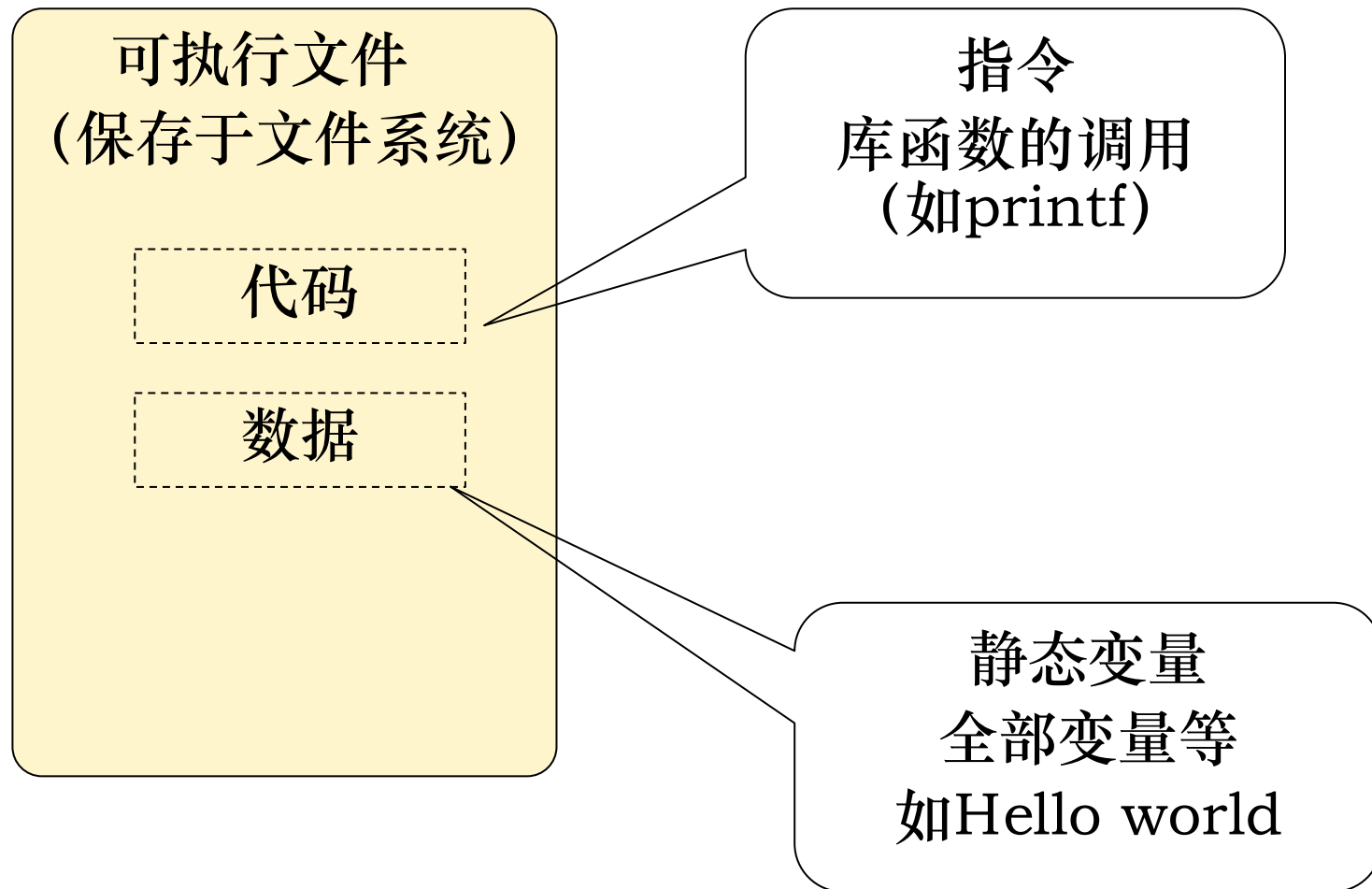
目标文件中，无类型、无符号、无数据结构

gcc默认情况下中间文件不保留，直接生成目标文件  
gcc -s 参数可以保留编译过程中的中间文件，比如as文件



## 2.2 程序的结构

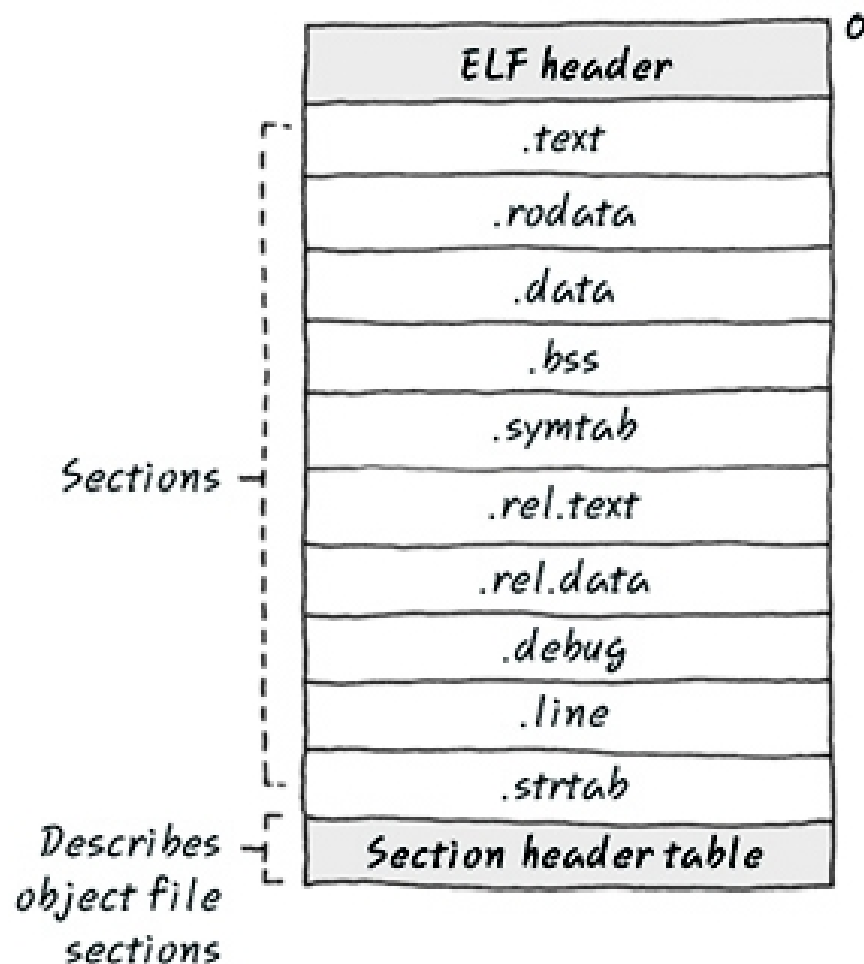
### ■ 可执行文件的格式/结构是怎么样的？



# 程序的结构

## ■ ELF文件格式

- 头信息
- 代码节
- 数据节
- 重定位节
- 符号表
- 调试信息等



# 三类目标文件

- 可重定位目标文件 (.o)
  - 其代码和数据可和其他可重定位文件合并为可执行文件
    - 每个.o 文件由对应的.c文件生成
    - 每个.o文件代码和数据地址都从0开始
- 可执行目标文件 (默认为a.out)
  - 包含的代码和数据可以被直接复制到内存并被执行
  - 代码和数据地址为虚拟地址空间中的地址
- 共享的目标文件 (.so)
  - 特殊的可重定位目标文件，能在装入或运行时被装入到内存并自动被链接，称为共享库文件
  - Windows 中称其为 *Dynamic Link Libraries* (DLLs)

# 目标文件的格式

## ■ 标准的几种目标文件格式

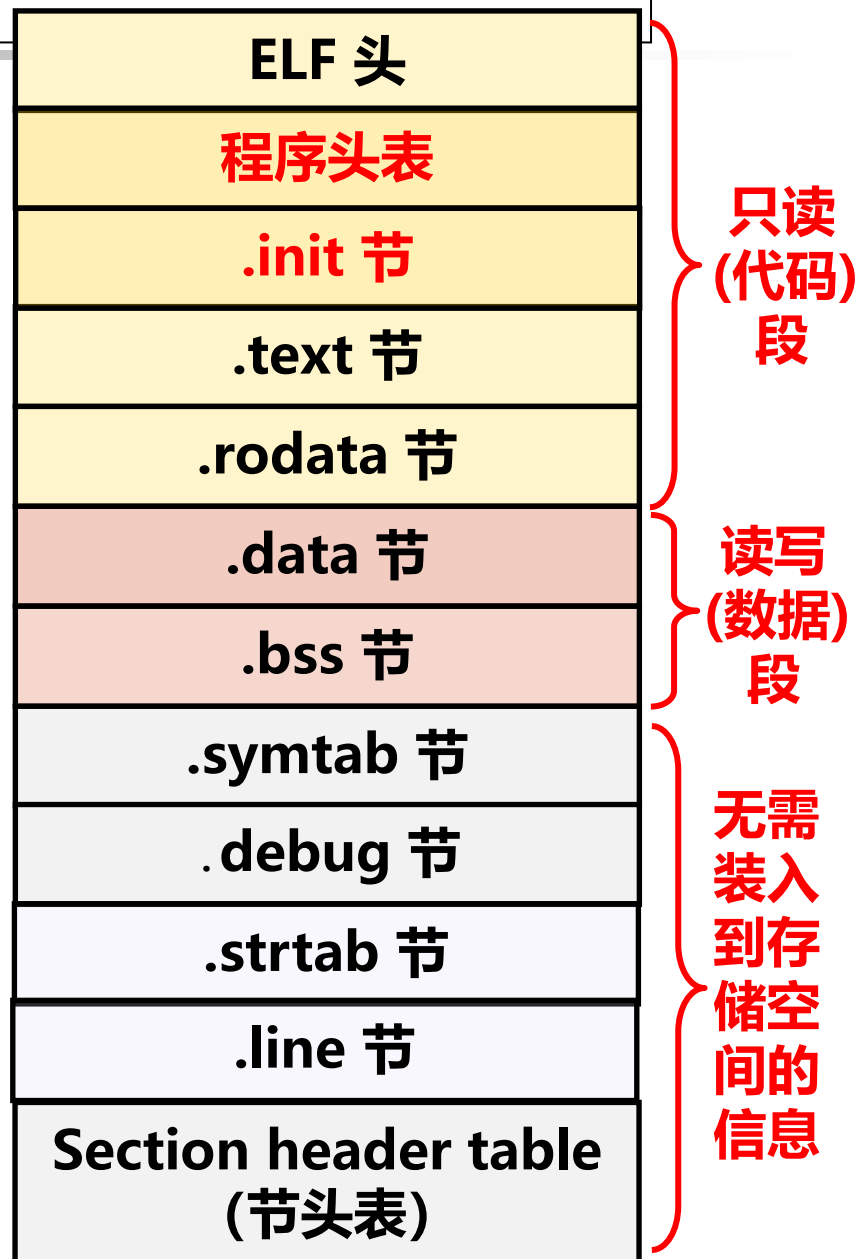
- DOS操作系统（最简单）：**COM格式**，文件中仅包含代码和数据，且被加载到固定位置
- System V UNIX早期版本：**COFF格式**，文件中不仅包含代码和数据，还包含重定位信息、调试信息、符号表等其他信息，由一组严格定义的数据结构序列组成
- Windows：**PE格式**（COFF的变种），称为可移植可执行（Portable Executable，简称PE）
- Linux等类UNIX：**ELF格式**（COFF的变种），称为可执行可链接（Executable and Linkable Format，简称ELF）

# 目标文件

- 目标文件并不能直接被OS装载运行
- 可重定位目标文件 (.o)
  - 其代码和数据可和其他可重定位文件合并为可执行文件
    - 每个.o文件代码和数据地址都从0开始
- 可执行目标文件 (默认为a.out)
  - 代码和数据地址为虚拟地址空间中的地址

# 可执行目标文件格式

- 与可重定位文件稍有不同：
  - ELF头中字段e\_entry给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0
  - 多一个程序头表，也称段头表 (segment header table)，是一个结构数组
  - 多一个.init节，用于定义\_init函数，该函数用来进行可执行目标文件开始执行时的初始化工作
  - 少两个.rel节 (无需重定位)



# ELF头信息举例

```
$ readelf -h main
```

ELF Header:

Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00

Class: ELF32

Data: 2's complement, little endian

Version: 1 (current)

OS/ABI: UNIX - System V

ABI Version: 0

Type: EXEC (Executable file)

Machine: Intel 80386

Version: 0x1

Entry point address: x8048580

Start of program headers: 52 (bytes into file)

Start of section headers: 3232 (bytes into file)

Flags: 0x0

Size of this header: 52 (bytes)

Size of program headers: 32 (bytes)

Number of program headers: 8

Size of section headers: 40 (bytes)

Number of section headers: 29

Section header string table index: 26

ELF 头
程序头表
.init 节
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.debug 节
.strtab 节
.line 节
Section header table (节头表)

# 可执行文件中的程序头表

- 程序头表描述可执行文件中的节与虚拟空间中的存储段之间的映射关系
- 一个表项（32B）说明虚拟地址空间中一个连续的段或一个特殊的节

```
typedef struct {  
    Elf32_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;
```



# 可执行文件中的程序头表

Program Headers:

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

- 有8个表项，其中两个为可装入段（即Type=LOAD）

# 可执行文件中的程序头表

Program Headers:

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

**第一可装入段：**第0x00000~0x004d3字节（包括ELF头、程序头表、.init、.text和.rodata节），映射到虚拟地址0x8048000开始长度为0x4d4字节的区域，按0x1000对齐，具有只读/执行权限（Flg=RE），是只读代码段。

**第二可装入段：**第0x000f0c开始长度为0x108字节的数据节，映射到虚拟地址0x8049f0c开始长度为0x110字节的存储区域，按0x1000对齐，具有可读可写权限（Flg=RW），是可读写数据段。

# 可执行文件中的节头表

Section Headers:

一共是31个段表 20135322

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	08048154	000154	000013	00	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	08048168	000168	000020	00	A	0	0	4
[ 3]	.note.gnu.build-i	NOTE	08048188	000188	000024	00	A	0	0	4
[ 4]	.gnu.hash	GNU_HASH	080481ac	0001ac	000020	04	A	5	0	4
[ 5]	.dynsym	DYNSYM	080481cc	0001cc	000050	10	A	6	1	4
[ 6]	.dynstr	STRTAB	0804821c	00021c	00004a	00	A	0	0	1
[ 7]	.gnu.version	VERSYM	08048266	000266	00000a	02	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	08048270	000270	000020	00	A	6	1	4
[ 9]	.rel.dyn	REL	08048290	000290	000008	08	A	5	0	4
[10]	.rel.plt	REL	08048298	000298	000010	08	AI	5	24	4
[11]	.init	PROGBITS	080482a8	0002a8	000023	00	AX	0	0	4
[12]	.plt	PROGBITS	080482d0	0002d0	000030	04	AX	0	0	16
[13]	.plt.got	PROGBITS	08048300	000300	000008	00	AX	0	0	8
[14]	.text	PROGBITS	08048310	000310	000192	00	AX	0	0	16
[15]	.fini	PROGBITS	080484a4	0004a4	000014	00	AX	0	0	4
[16]	.rodata	PROGBITS	080484b8	0004b8	00000e	00	A	0	0	4
[17]	.eh_frame_hdr	PROGBITS	080484c8	0004c8	00002c	00	A	0	0	4
[18]	.eh_frame	PROGBITS	080484f4	0004f4	0000cc	00	A	0	0	4
[19]	.init_array	INIT_ARRAY	08049f08	000f08	000004	00	WA	0	0	4
[20]	.fini_array	FINI_ARRAY	08049f0c	000f0c	000004	00	WA	0	0	4
[21]	.jcr	PROGBITS	08049f10	000f10	000004	00	WA	0	0	4
[22]	.dynamic	DYNAMIC	08049f14	000f14	0000e8	08	WA	6	0	4
[23]	.got	PROGBITS	08049ffc	000ffc	000004	04	WA	0	0	4
[24]	.got.plt	PROGBITS	0804a000	001000	000014	04	WA	0	0	4
[25]	.data	PROGBITS	0804a014	001014	000008	00	WA	0	0	4
[26]	.bss	NOBITS	0804a01c	00101c	000004	00	WA	0	0	1
[27]	.comment	PROGBITS	00000000	00101c	00002d	01	MS	0	0	1
[28]	.shstrtab	STRTAB	00000000	0016c9	00010a	00		0	0	1
[29]	.symtab	SYMTAB	00000000	00104c	000450	10		30	47	4
[30]	.strtab	STRTAB	00000000	00149c	00022d	00		0	0	1

---

# 程序的装入

# 1、绝对装入方式(Cont.)

```
org    07c00h                ; 告诉编译器程序加载到7c00处（见下面解释）
mov     ax, cs
mov     ds, ax
mov     es, ax
call    DispStr               ; 调用显示字符串例程
jmp     $                     ; 无限循环
```

DispStr:

```
mov     ax, BootMessage


---


mov     bp, ax                ; ES:BP = 串地址
mov     cx, 16                ; CX = 串长度
mov     ax, 01301h            ; AH = 13, AL = 01h
mov     bx, 000ch             ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
mov     dl, 0
int     10h                   ; 10h 号中断（这是BIOS中断，功能号AH=13表示“从指定位置起显示字符串”，详见附件）
```

ret

```
BootMessage:                db    "Hello, OS world!"
times 510-($-$$)            db    0        ; 填充剩下的空间，使生成的二进制代码恰好为512字节
dw      0xaa55               ; 结束标志
```

# 1、绝对装入方式(Cont.)

```
org      07c00h          ; 告诉编译器
mov      ax, cs
mov      ds, ax
mov      es, ax
call     DispStr          ; 调用显示
jmp      $                ; 无限循环
```

DispStr:

```
    mov     ax, BootMessage
    mov     bp, ax          ; ES:BP =
    mov     cx, 16          ; CX = 串
    mov     ax, 01301h      ; AH = 13
    mov     bx, 000ch        ; 页号为0
    mov     dl, 0
    int     10h             ; 10h 号中
```

串”，详见附件）

```
    ret
```

```
BootMessage:      db      "Hello, OS world!"
times 510-($-$$)  db      0          ; 填充剩下
dw      0xaa55      ; 结束标志
```

//ndisasm -o 0x7c00 boot.bin >> disboot.asm

//下面是反汇编boot.bin得到的disboot.asm文件:

//1. 程序框架

00007C00	8CC8	mov ax,cs
00007C02	8ED8	mov ds,ax
00007C04	8EC0	mov es,ax
00007C06	E80200	call word 0x7c0b
00007C09	EBFE	jmp short 0x7c09

//2. 显示字符串子例程

00007C0B	B81E7C	mov ax,0x7c1e
00007C0E	89C5	mov bp,ax
00007C10	B91000	mov cx,0x10
00007C13	B80113	mov ax,0x1301
00007C16	BB0C00	mov bx,0xc
00007C19	B200	mov dl,0x0
00007C1B	CD10	int 0x10
00007C1D	C3	ret

# 现代OS中的程序生成与装入

```
#include <stdio.h>

char str[16] = "Hello!";
int a = 1111;
int b = 2222;
void main()
{
    printf("A is %d at %x\n", a, &a);
    printf("B is %d at %x\n", b, &b);
    printf("Str is %s at %x\n", str, &str);
}
```



```
leizhao@ubuntu:~/Desktop$ ./printf_test
A is 1111 at 804a030
B is 2222 at 804a034
Str is Hello! at 804a020
leizhao@ubuntu:~/Desktop$
```

# 现代OS中的程序生成与装入

```
#include <stdio.h>

char str[16] = "A is B is Str is ";
int a = 1111;
int b = 2222;
void main()
{
    printf("A is %d\n", a);
    printf("B is %d\n", b);
    printf("Str is %s\n", str);
}
```

0804841d <main>:

804841d: 55	push	%ebp
804841e: 89 e5	mov	%esp, %ebp
8048420: 83 e4 f0	and	\$0xfffffffff0, %esp
8048423: 83 ec 10	sub	\$0x10, %esp
8048426: a1 30 a0 04 08	mov	0x804a030, %eax
804842b: c7 44 24 08 30 a0 04	movl	\$0x804a030, 0x8(%esp)
8048432: 08		
8048433: 89 44 24 04	mov	%eax, 0x4(%esp)
8048437: c7 04 24 10 85 04 08	movl	\$0x8048510, (%esp)
804843e: e8 ad fe ff ff	call	80482f0 <printf@plt>
8048443: a1 34 a0 04 08	mov	0x804a034, %eax
8048448: c7 44 24 08 34 a0 04	movl	\$0x804a034, 0x8(%esp)
804844f: 08		
8048450: 89 44 24 04	mov	%eax, 0x4(%esp)
8048454: c7 04 24 1f 85 04 08	movl	\$0x804851f, (%esp)
804845b: e8 90 fe ff ff	call	80482f0 <printf@plt>
8048460: c7 44 24 08 20 a0 04	movl	\$0x804a020, 0x8(%esp)
8048467: 08		
8048468: c7 44 24 04 20 a0 04	movl	\$0x804a020, 0x4(%esp)
804846f: 08		
8048470: c7 04 24 2e 85 04 08	movl	\$0x804852e, (%esp)
8048477: e8 74 fe ff ff	call	80482f0 <printf@plt>
804847c: c9	leave	
804847d: c3	ret	



# 现代OS中的程序生成与装入

```
#include <stdio.h>

char str[16] = "Hello!";
int a = 1111;
int b = 2222;
void main()
{
    printf("A is %d at %x\n", a, &a);
    printf("B is %d at %x\n", b, &b);
    printf("Str is %s at %x\n", str, &str);
}
```

```
leizhao@ubuntu:~/Desktop$ ./printf_test
A is 1111 at 804a030
B is 2222 at 804a034
Str is Hello! at 804a020
```

```
#include <stdio.h>
char buf[16] = "World!";
int c = 3333;
int d = 4444;
void main()
{
    c = c+d;
    c = c-d;
    printf("C is %d at %x\n", c, &c);
    printf("D is %d at %x\n", d, &d);
    printf("Buf is %s at %x\n", buf, &buf);
}
```

```
leizhao@ubuntu:~/Desktop$ ./printf_test2
C is 3333 at 804a030
D is 4444 at 804a034
Buf is World! at 804a020
```

相同“地址”的  
数据不同???

# 现代OS中的程序生成与装入

```
0804841d <main>:
804841d: 55                push    %ebp
804841e: 89 e5             mov     %esp,%ebp
8048420: 83 e4 f0          and     $0xfffffffff0,%esp
8048423: 83 ec 10          sub     $0x10,%esp
8048426: a1 30 a0 04 08    mov     0x804a030,%eax
804842b: c7 44 24 08 30 a0 04 movl    $0x804a030,0x8(%esp)
8048432: 08
8048433: 89 44 24 04       mov     %eax,0x4(%esp)
8048437: c7 04 24 10 85 04 08 movl    $0x8048510,(%esp)
804843e: e8 ad fe ff ff    call    80482f0 <printf@plt>
```

```
0804841d <main>:
804841d: 55                push    %ebp
804841e: 89 e5             mov     %esp,%ebp
8048420: 83 e4 f0          and     $0xfffffffff0,%esp
8048423: 83 ec 10          sub     $0x10,%esp
8048426: 8b 15 30 a0 04 08 mov     0x804a030,%edx
804842c: a1 34 a0 04 08    mov     0x804a034,%eax
8048431: 01 d0             add     %edx,%eax
8048433: a3 30 a0 04 08    mov     %eax,0x804a030
8048437: 8b 15 30 a0 04 08 mov     0x804a030,%edx
804843d: 1 34 a0 04 08     mov     0x804a034,%eax
8048441: 9 c2             sub     %eax,%edx
8048443: d0             mov     %edx,%eax
8048445: 13 30 a0 04 08    mov     %eax,0x804a030
8048449: a1 30 a0 04 08    mov     0x804a030,%eax
804844b: c7 44 24 08 30 a0 04 movl    $0x804a030,0x8(%esp)
```

相同“地址”的  
指令也不同???

# 现代OS中的程序生成与装入

## ■ 我们的观察

- 两个进程并发执行
  - 执行各自的指令
  - 读取各自的数据
- 两个进程的内存空间
  - 同一地址的指令不同
  - 同一地址的数据也不同

## ■ 已知事实

- 硬件上，一个内存单元能同时存不同的数据??

地址假?



数据假?

# 现代OS中的程序生成与装入

## ■ 推理可知

- 指令中的内存地址应该为逻辑地址
  - 同一“地址”的指令和数据均不冲突
- OS负责将逻辑地址转变为物理地址
  - 硬件上，只能靠物理地址来寻址

## ■ 地址变换

---

# 程序的链接

## 2.3 程序的链接

### ■ 可重定位目标文件

- 可被链接（合并）生成可执行文件或**共享目标文件**
- **静态链接库文件**由若干个可重定位目标文件组成
- 包含**重定位信息**（指出哪些符号引用处需要重定位）

### ■ 可执行文件

- 将多个目标文件内具相同访问属性的节合并，并说明每个段的属性，如：在可执行文件中的位移、大小、在虚拟空间中的位置、对齐方式、访问属性等
- 定义的所有变量和函数已有确定的虚拟地址
- 符号引用处已被重定位，以指向所引用的定义符号
- 可被CPU直接执行，指令地址和指令给出的操作数地址都是虚拟地址

# 链接器的由来

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



**穿孔表示0，未穿孔为1**

**假设：0010-jmp**

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

若指令偏移发生变化（如删减指令），  
则程序员需重新计算jmp指令的目标地址（**重定位**），然后重新打孔。

# 链接器的由来

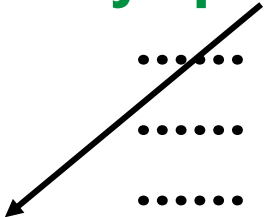
- 用**符号**表示跳转位置和变量位置
- 汇编语言出现
  - 用助记符表示操作码
  - 用**符号**表示位置
  - 用助记符表示寄存器

■ .....

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

add B  
jmp L0

.....  
.....  
.....  
L0: sub C  
.....





# 链接器的由来

## ■ 更高级编程语言出现

- 程序越来越复杂，需多人开发不同的程序模块
- 子程序（函数）起始地址和变量起始地址是符号定义（definition）
- **调用**子程序（函数或过程）和**使用**变量即是符号的引用（reference）
- 一个模块定义的符号可以被另一个模块引用
- 最终须链接（即合并），合并时须在符号引用处填入定义处的地址

# 使用链接的好处

## ■ 模块化

- 可以将程序分成很多源程序文件
- 可构建公共函数库，如libc

## ■ 高效

- 独立编译
  - 只需重新编译被修改的源程序文件，然后重新连接
- 无需包含共享库的所有代码
  - 源文件无需包含共享库函数的源码，只要直接调用
  - 可执行文件和运行时的内存中只需包含所调用函数的代码，而不需要包含整个共享库

# 一个C语言程序举例

## main.c

```
int buf[2] = {1, 2};  
void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

## swap.c

```
extern int buf[];  
int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
{  
    int temp;  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

哪些是**符号定义**? 哪些是**符号的引用**?

# 一个C语言程序举例

哪些是**符号定义**? 哪些是符号的引用?

## main.c

```
int buf[2] = {1, 2};  
void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

## swap.c

```
extern int buf[];  
int *bufp0 = &buf[0];  
static int *bufp1;  
void swap()  
{  
    int temp;  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

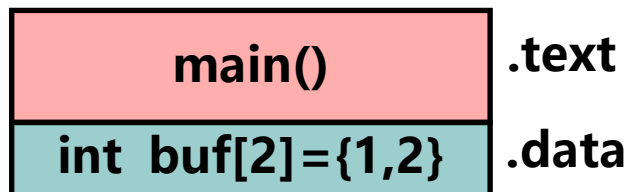
局部变量**temp**分配在栈中，不会在过程外被引用，因此不是符号定义

# 链接过程的本质

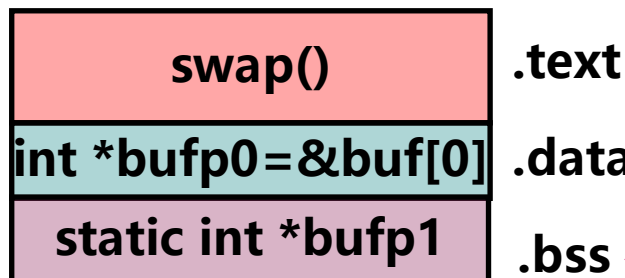
- 链接：合并相同的“节”

## 可重定位目标文件

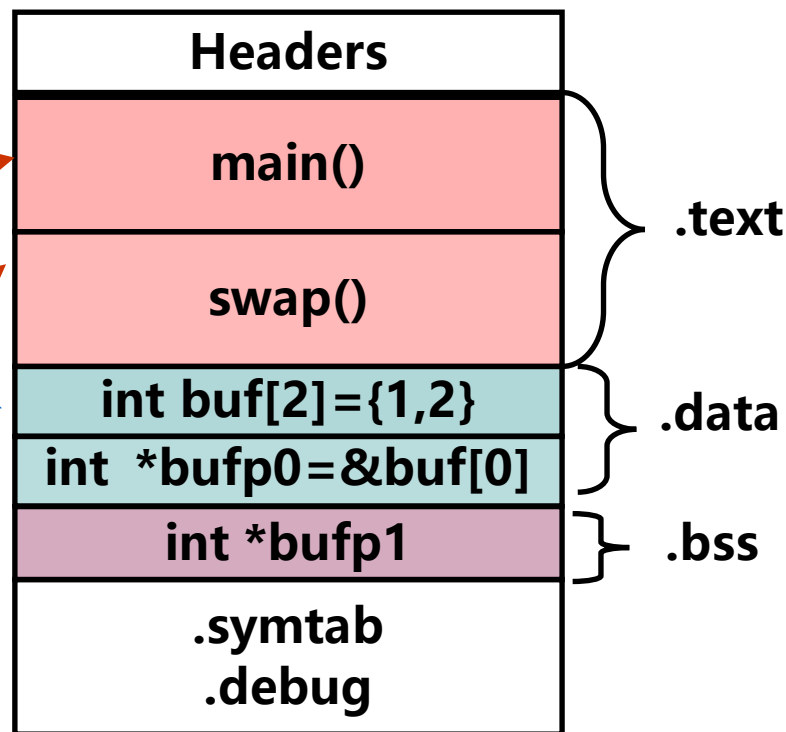
main.o



swap.o



## 可执行目标文件



# 目标文件

```
/* main.c */
int add(int, int);
int main( )
{
    return add(20, 13);
}
```

```
/* test.c */
int add(int i, int j)
{
    int x = i + j;
    return x;
}
```

00000000

0:

<add>:

**objdump -d test.o**

1:

3:

6:

9:

c:

f:

12:

15:

16:

55

89 e5

83 ec 10

8b 45 0c

8b 55 08

8d 04 02

89 45 fc

8b 45 fc

c9

c3

push %ebp

mov %esp, %ebp

sub \$0x10, %esp

mov 0xc(%ebp), %eax

mov 0x8(%ebp), %edx

lea (%edx,%eax,1), %eax

mov %eax, -0x4(%ebp)

mov -0x4(%ebp), %eax

leave

ret

080483d4

80483d4:

80483d5:

80483d7:

80483da:

80483dd:

80483e0:

80483e3:

80483e6:

80483e9:

80483ea:

<add>:

**objdump -d test**

55

89 e5

83 ec 10

8b 45 0c

8b 55 08

8d 04 02

89 45 fc

8b 45 fc

c9

c3

push %ebp

mov %esp, %ebp

sub \$0x10, %esp

mov 0xc(%ebp), %eax

mov 0x8(%ebp), %edx

lea (%edx,%eax,1), %eax

mov %eax, -0x4(%ebp)

mov -0x4(%ebp), %eax

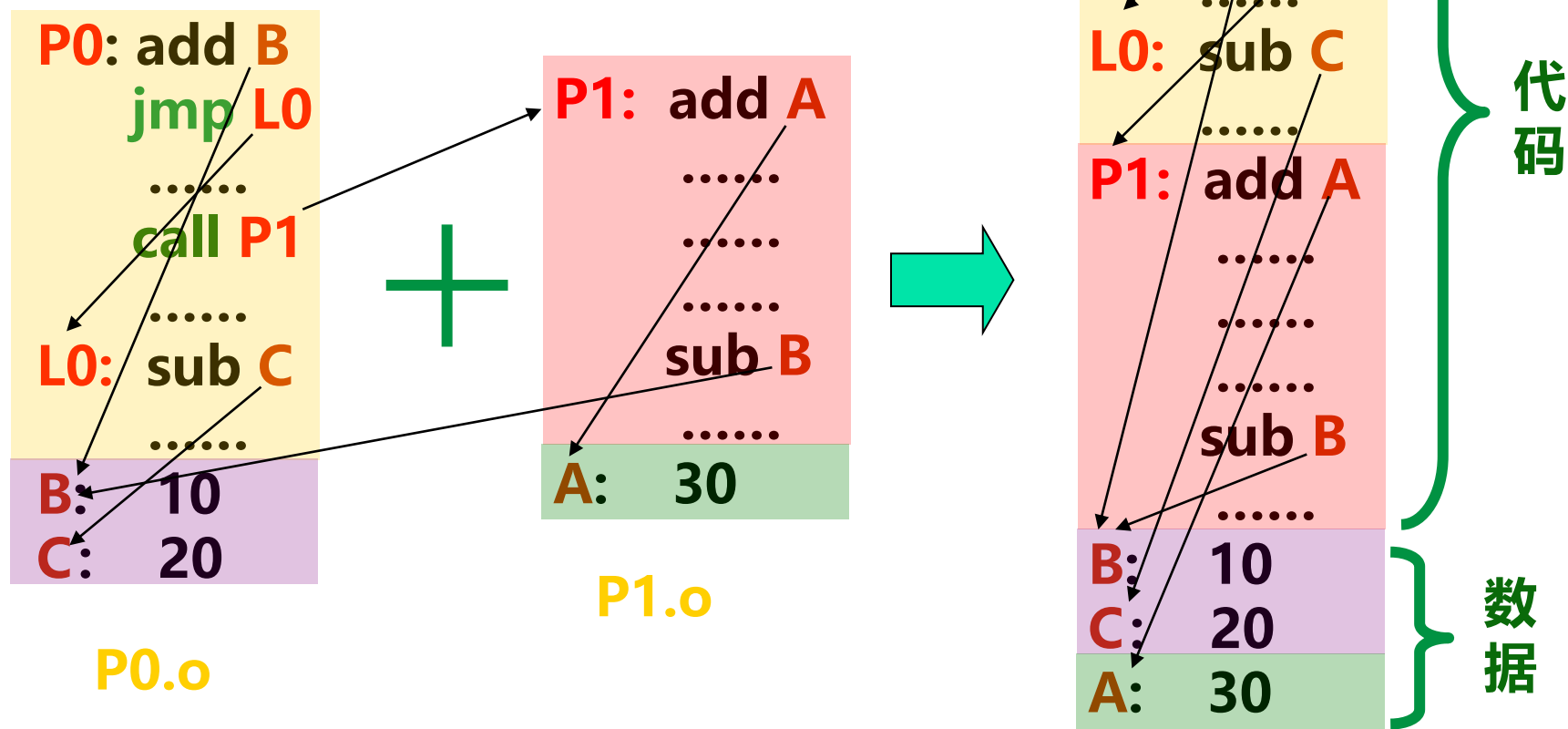
leave

ret

# 链接操作的步骤

- 1) 确定标号引用关系 (符号解析)
- 2) 合并相关.o文件
- 3) 确定每个标号的地址
- 4) 在指令中填入新地址

重定位



# 链接操作的步骤

## ■ Step 1. 符号解析 (Symbol resolution)

- 程序中有定义和引用的符号 (包括变量和函数等)
  - `void swap() {...} /* 定义符号swap */`
  - `swap(); /* 引用符号swap */`
  - `int *xp = &x; /* 定义符号 xp, 引用符号 x */`
- 编译器将**定义的符号**存放在一个**符号表** (symbol table) 中.
  - 符号表是一个结构数组
  - 每个表项包含符号名、**长度和位置**等信息
- 链接器将每个**符号的引用**都与一个确定的**符号定义**建立关联

## ■ Step 2. 重定位

- 将多个代码段与数据段分别**合并为**一个单独的代码段和数据段
- 计算每个定义的符号在虚拟地址空间中的**绝对地址**
- 将可执行文件中符号引用处的地址**修改为重定位后的地址信息**



# 符号和符号解析

- Global symbols（模块内部定义的全局符号）
  - 由模块m定义并能被其他模块引用的符号。例如，非static 函数和非static的全局变量（指不带static的全局变量）
  - 如，main.c 中的全局变量名buf
- External symbols（外部定义的全局符号）
  - 由其他模块定义并被模块m引用的全局符号
  - 如，main.c 中的函数名swap
- Local symbols（本模块的局部符号）
  - 仅由模块m定义和引用的本地符号。例如，在模块m中定义的带static的函数和全局变量
  - 如，swap.c 中的static变量名bufp1
  - 链接器局部符号不是指程序中的局部变量（分配在栈中的临时性变量），链接器不关心这种局部变量

# 符号和符号解析

## main.c

```
int buf[2] = {1, 2};  
extern void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

## swap.c

```
extern int buf[];  
  
int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
{  
    int temp;  
  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

哪些是全局符号？ 哪些是外部符号？ 哪些是局部符号？

# 目标文件中的符号表

- .symtab 节记录符号表信息，是一个结构数组
- 符号表（symtab）中每个条目的结构如下：

```
typedef struct {  
    int  name; /*符号对应字符串在strtab节中的偏移量，序号*/  
    int  value; /*在对应节中的偏移量，可执行文件中是虚拟地址*/  
    int  size; /*符号对应目标所占字节数：函数大小或变量大小*/  
    char type: 4, /*符号对应目标的类型：数据、函数、源文件、节*/  
        binding: 4; /*符号类别：全局符号、局部符号、弱符号*/  
    char reserved;  
    char section; /*符号对应目标所在的节，或其他情况*/  
} Elf_Symbol;
```

# 目标文件中的符号表

## ■ main.o中的符号表中最后三个条目（共10个）

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	Data	Global	0	3	buf
9:	0	18	Func	Global	0	1	main
10:	0	0	Notype	Global	0	UND	swap

Buf: main.o中第3节 (.data) 偏移为0的符号, 全局变量, 占8B ( ? ) ;  
main是第1节 (.text) 偏移为0的符号, 全局函数, 占33B;  
swap是main.o中未定义全局 (在其他模块定义) 符号, 类型和大小未知

# 符号解析 (Symbol Resolution)

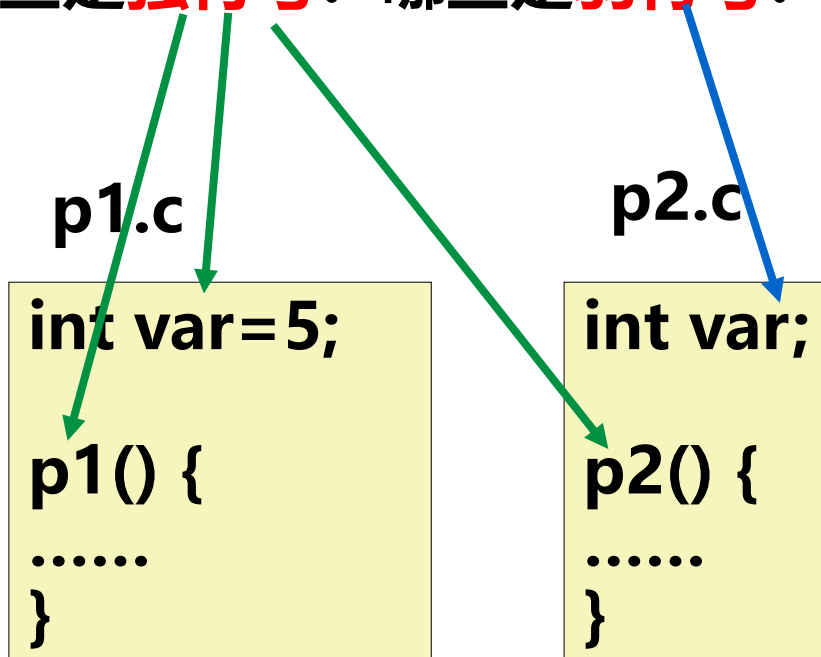
- 目的：将每个模块中**引用的符号**与某个目标模块中的**定义符号**建立关联。
- 每个**定义符号**在代码段或数据段中都被分配了存储空间，将**引用符号**与**定义符号**建立关联后，就可在重定位时将**引用符号的地址**重定位为**相关联的定义符号的地址**。
- 定义符号的值就是目标所在的首地址
- **本地符号**在本模块内定义并引用，因此，其解析较简单，只要与本模块内唯一的定义符号关联即可。
- **全局符号**（外部定义的、内部定义的）的解析涉及多个模块，故较复杂

# 全局符号的符号解析

## ■ 全局符号的强/弱特性

- 函数名和已初始化的全局变量名是**强符号**
- 未初始化的全局变量名是**弱符号**

以下符号哪些是**强符号**？ 哪些是**弱符号**？



# 全局符号的符号解析

以下符号哪些是**强符号**？ 哪些是**弱符号**？

main.c

```
int buf[2] = {1, 2};  
void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

此处为引用

swap.c

```
extern int buf[];  
int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
{  
    int temp;  
  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

局部变量

本地局部符号

# 链接器对符号的解析规则

## ■ 多重定义符号的处理规则

Rule 1: 强符号不能多次定义

- 强符号只能被定义一次，否则链接错误

Rule 2: 若一个符号被定义为一次强符号和多次弱符号，则按强定义为准

- 对弱符号的引用被解析为其强定义符号

Rule 3: 若有多个弱符号定义，则任选其中一个

- 使用命令 `gcc -fno-common` 链接时，会告诉链接器在遇到多个弱定义的全局符号时输出一条警告信息。



# 多重定义全局符号的问题

- 尽量避免使用全局变量
- 一定需要用的话，就按以下规则使用
  - 尽量使用本地变量（static）
  - 全局变量要赋初值
  - 外部全局变量要使用extern

**多重定义全局变量会造成一些意想不到的错误，而且是默默发生的，编译系统不会警告，并会在程序执行很久后才能表现出来，且远离错误引发处。特别是在一个具有几百个模块的大型软件中，这类错误很难修正。**

**大部分程序员并不了解链接器如何工作，因而养成良好的编程习惯是非常重要的。**

## 2.4 重定位

符号解析完成后，可进行重定位工作，分三步

### ■ 合并相同的节

- 将所有目标模块中相同的节合并成新节

例如，所有.text节合并作为可执行文件中的.text节

### ■ 对定义符号进行重定位（确定地址）

- 确定新节中所有定义符号在虚拟地址空间中的地址

例如，为函数确定首地址，进而确定每条指令的地址，为变量确定首地址

- 完成这一步后，每条指令和每个全局变量都可确定地址

### ■ 对引用符号进行重定位（确定地址）

- 修改.text节和.data节中对每个符号的引用（地址）

需要用到在.rel\_data和.rel\_text节中保存的重定位信息

# 重定位信息

- 汇编器遇到引用时，生成一个重定位条目
- 数据引用的重定位条目在.rel\_data节中
- 指令引用的重定位条目在.rel\_text节中
- ELF中重定位条目格式如下：

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

- IA-32有两种最基本的重定位类型
  - R\_386\_32: 绝对地址
  - R\_386\_PC32: PC相对地址

# 重定位操作举例

## main.c

```
int buf[2] = {1, 2};  
void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

## swap.c

```
extern int buf[];  
int *bufp0 = &buf[0];  
static int *bufp1;  
void swap()  
{  
    int temp;  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

在main.o和swap.o的**重定位节(.rel.text、.rel.data)**中有**重定位信息**，反映符号引用的位置、绑定的定义符号名、重定位类型

# main.o重定位前

main的定义在.text  
节中偏移为0处开始,  
占0x12B。

Disassembly of section .text: main.o

00000000 <main>:

0:	55	push	%ebp
1:	89 e5	mov	%esp,%ebp
3:	83 e4 f0	and	\$0xffffffff0,%esp
6:	e8 <span style="border: 1px solid black; padding: 2px;">fc ff ff ff</span>	call	7 <main+0x7>
			<b>7: R_386_PC32 swap</b>
b:	b8 00 00 00 00	mov	\$0x0,%eax
10:	c9	leave	
11:	c3	ret	

在rel\_text节中的重定位条目为:

r\_offset=0x7, r\_sym=10, r\_type=R\_386\_PC32,  
dump出来后为 **"7: R\_386\_PC32 swap"**

r\_sym=10说明引用的是swap!

# main.o中的符号表

## ■ main.o中的符号表中最后三个条目

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	Data	Global	0	3	buf
9:	0	18	Func	Global	0	1	main
10:	0	0	Notype	Global	0	UND	swap

swap是main.o的符号表中第10项，是未定义符号，类型和大小未知，并是全局符号，故在其他模块中定义。

# R\_386\_PC32的重定位方式

PC相对地址方式下，重定位值计算公式为：

$$\text{ADDR}(r\_sym) - ( ( \text{ADDR}(.text) + r\_offset ) - \text{init} )$$

## ■ 假定：

- 可执行文件中main函数对应机器代码从0x8048380开始
- swap紧跟main后，其机器代码首地址按4字节边界对齐

## ■ 则swap起始地址为多少？

- $0x8048380 + 0x12 = 0x8048392$ ，对齐后为0x8048394

## ■ 则重定位后call指令的机器代码是什么？

- 转移目标地址 = PC + 偏移地址

PC =  $0x8048380 + 0x07 - \text{init}$  (init: 指令长度的修正)

PC =  $0x8048380 + 0x07 - (-4) = 0x804838b$

- 重定位值 = 转移目标地址 - PC =  $0x8048394 - 0x804838b = 0x9$
- call指令的机器代码为 "e8 09 00 00 00"

# R\_386\_32的重定位方式

## Disassembly of section .data:

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

buf定义在.data节  
偏移0处, 占8B,  
无需重定位的符号

main.c

```
int buf[2]={1,2};  
  
int main()  
.....
```

## swap.o中.data和.rel.data节内容

## Disassembly of section .data:

```
00000000 <bufp0>:  
0: 00 00 00 00  
0: R_386_32 buf
```

bufp0定义  
在.data节中  
偏移为0处,  
占4B, 初值  
为0x0

swap.c

```
extern int buf[];  
  
int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
.....
```

重定位节.rel.data中有一个重定位表项:  $r\_offset=0x0$ ,  $r\_sym=9$ ,  
 $r\_type=R\_386\_32$   
 $r\_sym=9$ 说明引用的是buf!

OBJDUMP工具解释后显示为 "0: R\_386\_32 buf"



# swap.o中的符号表

## ■ swap.o中的符号表中最后4个条目

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

buf是swap.o的符号表中第9项，是未定义符号，类型和大小未知，并是全局符号，故在其他模块中定义。

# R\_386\_32的重定位方式

## ■ 假定：

- buf在运行时的存储地址ADDR(buf)=0x8049620

## ■ 则重定位后，bufp0的地址及内容变为什么？

- buf和bufp0同属于.data节，故在可执行文件中它们被合并
- bufp0紧接在buf后，故地址为0x8049620+8= 0x8049628
- 因是R\_386\_32方式，故**bufp0内容**为buf的绝对地址0x8049620，即 “20 96 04 08”

### Disassembly of section .data:

08049620 <buf>:

**8049620:** 01 00 00 00 02 00 00 00

08049628 <bufp0>:

**8049628:** **20 96 04 08**

# swap.o重定位

```
extern int buf[];
```

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

```
void swap() swap.c  
{  
    int temp;
```

```
    bufp1 = &buf[1];
```

```
    temp = *bufp0;
```

```
    *bufp0 = *bufp1;
```

```
    *bufp1 = temp;
```

共有6处需要重定位

划红线处: 8、c、

11、1b、21、2a

Disassembly of section .text:

00000000 <swap>:

```
0: 55          push  %ebp  
1: 89 e5       mov   %esp,%ebp  
3: 83 ec 10    sub   $0x10,%esp
```

```
6: c7 05 00 00 00 00 04 movl  $0x4,0x0
```

```
d: 00 00 00
```

```
8: R_386_32    .bss  
c: R_386_32    buf
```

```
10: a1 00 00 00 00 mov  0x0,%eax
```

```
11: R_386_32    bufp0
```

```
15: 8b 00       mov  (%eax),%eax
```

```
17: 89 45 fc    mov  %eax,-0x4(%ebp)
```

```
1a: a1 00 00 00 00 mov  0x0,%eax
```

```
1b: R_386_32    bufp0
```

```
1f: 8b 15 00 00 00 00 mov  0x0,%edx
```

```
21: R_386_32    .bss
```

```
25: 8b 12       mov  (%edx),%edx
```

```
27: 89 10       mov  %edx,(%eax)
```

```
29: a1 00 00 00 00 mov  0x0,%eax
```

```
2a: R_386_32    .bss
```

```
2e: 8b 55 fc    mov  -0x4(%ebp),%edx
```

```
31: 89 10       mov  %edx,(%eax)
```

```
33: c9          leave
```

```
34: c3          ret
```

# swap.o重定位

buf和bufp0的地址分别是0x8049620和0x8049628

&buf[1](c处重定位值) 为0x8049620+0x4=0x8049624

bufp1的地址就是链接合并后.bss节的首地址, 假定为0x8049700

8 (bufp1): 00 97 04 08

c (&buf[1]): 24 96 04 08

11 (bufp0): 28 96 04 08

1b (bufp0): 28 96 04 08

21 (bufp1): 00 97 04 08

2a (bufp1): 00 97 04 08

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

6:	c7 05 00 00 00 00 04	movl \$0x4,0x0	
d:	00 00 00		
		8: R_386_32	.bss
		c: R_386_32	buf
10:	a1 00 00 00 00	mov 0x0,%eax	
11:		R_386_32	bufp0
15:	8b 00	mov (%eax),%eax	
17:	89 45 fc	mov %eax,-0x4(%ebp)	
1a:	a1 00 00 00 00	mov 0x0,%eax	
1b:		R_386_32	bufp0
1f:	8b 15 00 00 00 00	mov 0x0,%edx	
21:		R_386_32	.bss
25:	8b 12	mov (%edx),%edx	
27:	89 10	mov %edx,(%eax)	
29:	a1 00 00 00 00	mov 0x0,%eax	
2a:		R_386_32	.bss
2e:	8b 55 fc	mov -0x4(%ebp),%edx	
31:	89 10	mov %edx,(%eax)	

# 重定位后

08048380 <main>:

8048380: 55           push %ebp  
8048381: 89 e5       mov %esp,%ebp

8048383: 83 e5

8048386: e8 0

804838b: b8 0

8048390: c9

8048391: c3

8048392: 90

8048393: 90

08048394 <swap>:

8048394: 55           push %ebp  
8048395: 89 e5       mov %esp,%ebp  
8048397: 83 ec 10     sub \$0x10,%esp  
804839a: c7 05 00 97 04 08 24 mov \$0x8049624,0x8049700  
80483a1: 96 04 08  
80483a4: a1 28 96 04 08   mov 0x8049628,%eax  
80483a9: 8b 00       mov (%eax),%eax  
80483ab: 89 45 fc     mov %eax,-0x4(%ebp)  
80483ae: a1 28 96 04 08   mov 0x8049628,%eax  
80483b3: 8b 15 00 97 04 08 mov 0x8049700,%edx  
80493b9: 8b 12       mov (%edx),%edx  
80493bb: 89 10       mov %edx,(%eax)  
80493bd: a1 00 97 04 08   mov 0x8049700,%eax  
80493c2: 8b 55 fc     mov -0x4(%ebp),%edx  
80493c5: 89 10       mov %edx,(%eax)  
80493c7: c9           leave  
80493c8: c3           ret

假定每个函数  
要求4字节边界  
对齐,故填充两  
条nop指令

# 如何划分模块？

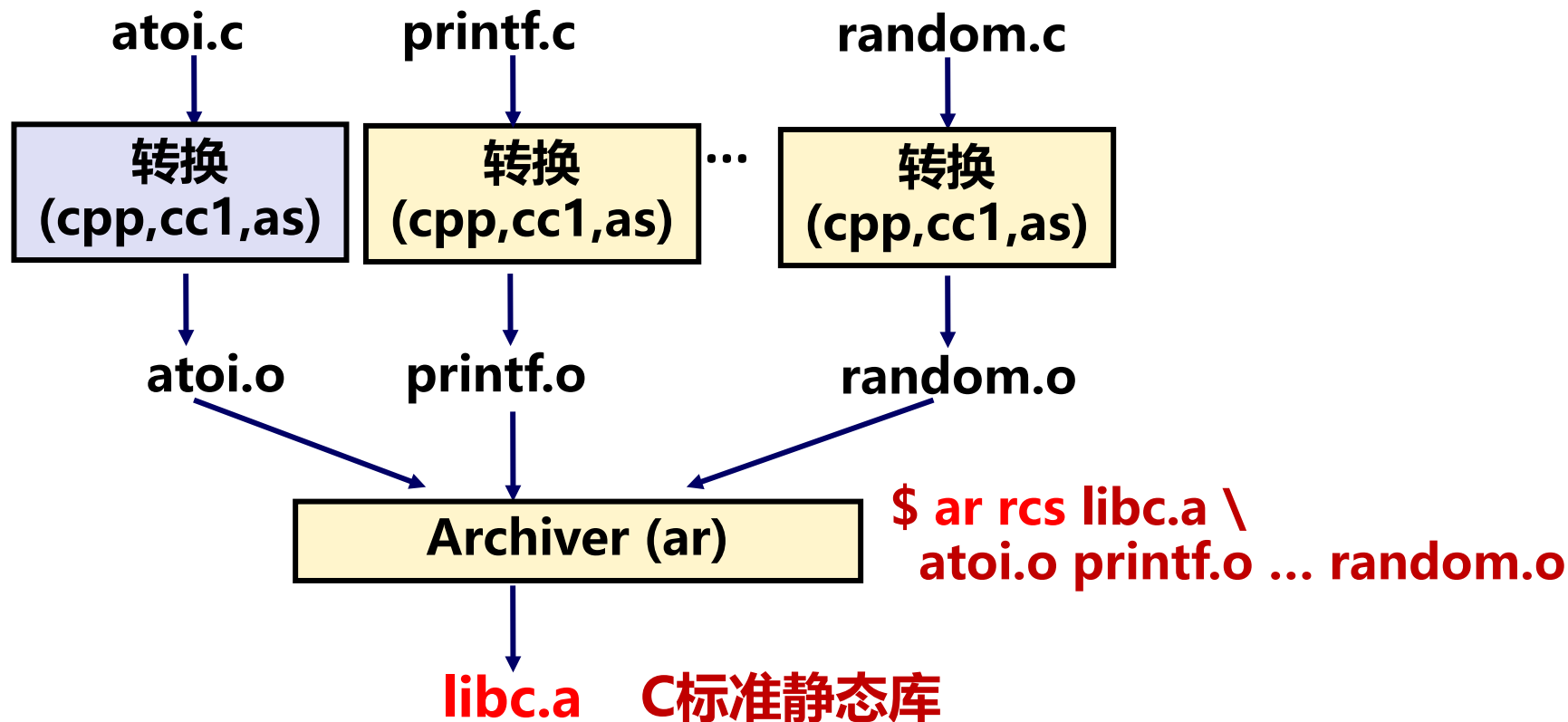
- 许多函数无需自己写，可使用共享库函数
  - 如数学库, 输入/输出库, 存储管理库, 字符串处理等
- 避免以下两种极端做法
  - 将所有函数都放在一个源文件中
    - 修改一个函数需要对所有函数重新编译
    - 时间和空间两方面的效率都不高
  - 一个源文件中仅包含一个函数
    - 需要程序员显式地进行链接
    - 效率高，但模块太多，故太繁琐

# 静态共享库

## ■ 静态库 (.a archive files)

- 将所有相关的目标模块 (.o) 打包为一个单独的库文件 (.a)，称为**静态库文件**，也称**存档文件** (archive)
- 增强了链接器功能，使其能通过查找一个或多个库文件中的符号来解析符号
- 在构建可执行文件时只需指定库文件名，链接器会自动到库中寻找那些应用程序用到的目标模块，并且**只把用到的模块从库中拷贝出来**
- 在gcc命令中无需明显指定C标准库libc.a(默认库)

# 静态库的创建



- Archiver (归档器) 允许增量更新, 只要重新编译需修改的源码并将其.o文件替换到静态库中。



# 常用静态库

- libc.a ( C标准库 )
  - 1392个目标文件 (大约8 MB)
  - 包含I/O、存储分配、信号处理、字符串处理、时间和日期、随机数生成、定点整数算术运算
- libm.a (the C math library)
  - 401 个目标文件 (大约 1 MB)
  - 浮点数算术运算(如sin, cos, tan, log, exp, sqrt, ...)

```
% ar -t /usr/lib/libc.a | sort
```

```
...  
fork.o  
...  
fprintf.o  
fpu_control.o  
fnputc.o
```

```
% ar -t /usr/lib/libm.a | sort
```

```
...  
e_acos.o  
e_acosf.o  
e_acosh.o  
e_acoshf.o  
e_acoshl.o
```

# 自定义一个静态库文件

举例：将myproc1.o和myproc2.o打包生成mylib.a

myproc1.c

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

myproc2.c

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

\$ gcc -c myproc1.c myproc2.c

\$ ar rcs mylib.a myproc1.o myproc2.o

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

\$ gcc -c main.c

\$ gcc -static -o myproc main.o ./mylib.a

调用关系：main→myfunc1→printf

问题：如何进行符号解析？

# 链接器中符号解析的全过程

**E** 将被合并以组成可执行文件的所有目标文件集合

**U** 当前所有未解析的引用符号的集合

**D** 当前所有定义符号的集合

- **开始E、U、D为空**，首先扫描main.o，把它加入E，同时把**myfun1加入U，main加入D**
- 接着扫描到mylib.a，将U中所有符号（本例中为myfunc1）与mylib.a中所有目标模块（myproc1.o和myproc2.o）依次匹配，发现在**myproc1.o中定义了myfunc1，故myproc1.o加入E，myfunc1从U转移到D。**
- 在myproc1.o中发现还有未解析符号**printf，将其加到U。**
- 不断在mylib.a的各模块上进行迭代以匹配U中的符号，直到U、D都不再变化。此时U中只有一个未解析符号printf，而D中有main和myfunc1
- 接着，扫描默认的库文件libc.a，发现其目标模块printf.o定义了printf，于是printf也从U移到D，并将printf.o加入E，同时把它定义的所有符号加入D，而所有未解析符号加入U。
- **处理完libc.a时，U一定是空的。**

# 链接器中符号解析的全过程

**\$ gcc -static -o myproc main.o ./mylib.a**

**main.c**

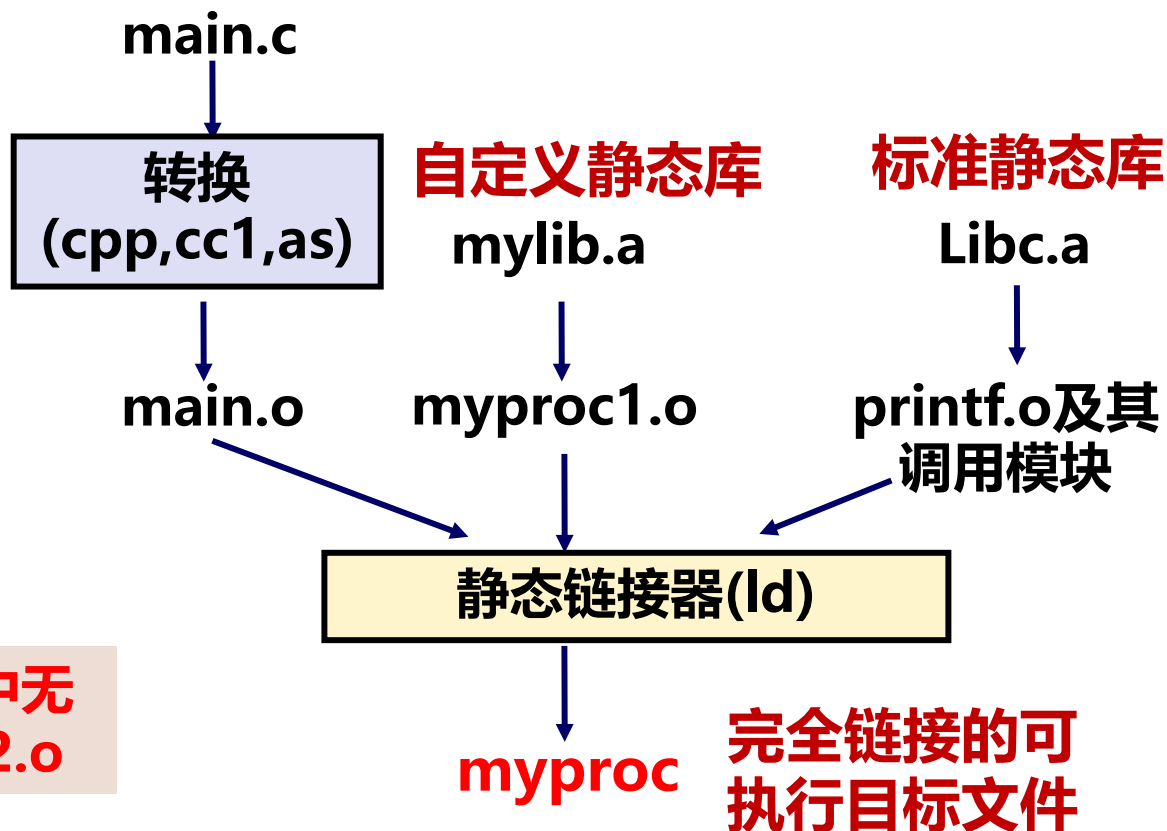
```
void myfunc1(viod);  
int main()  
{  
    myfunc1();  
    return 0;  
}
```

**解析结果:**

**E中有main.o、myproc1.o、printf.o及其调用的模块**

**D中有main、myproc1、printf及其引用的符号**

**注意: E中无  
myproc2.o**



# 使用静态库

- 链接器对外部引用的解析算法要点如下:
  - 按照命令行给出的顺序扫描.o 和.a 文件
  - 扫描期间将当前未解析的引用记录到一个列表U中
  - 每遇到一个新的.o 或 .a 中的模块, 都试图用其来解析U中的符号
  - 如果扫描到最后, U中还有未被解析的符号, 则发生错误
- 问题和对策
  - 能否正确解析与命令行给出的顺序有关
  - 好的做法: 将静态库放在命令行的最后

# 链接器中符号解析的全过程

- `$ gcc -static -o myproc ./mylib.a main.o ?`
  - 首先，扫描mylib，因是静态库，应根据其中是否存在U中未解析符号对应的定义符号来确定哪个.o被加入E。
  - 因为开始U为空，故mylib中两个.o模块都不被加入E中而被丢弃。
  - 然后，扫描main.o，将myfunc1加入U，直到最后它都不能被解析。
  - 因此，出现链接错误

# 链接顺序问题

- 假设调用关系如下：

func.o → libx.a 和 liby.a 中的函数

libx.a → libz.a 中的函数

libx.a 和 liby.a 之间、liby.a 和 libz.a 相互独立

则以下几个命令行都是可行的：

- gcc -static -o myfunc func.o libx.a liby.a libz.a
- gcc -static -o myfunc func.o liby.a libx.a libz.a
- gcc -static -o myfunc func.o libx.a libz.a liby.a

# 动态链接的共享库

## ■ 静态库有一些缺点：

- 库函数（如printf）被包含在每个运行进程的代码段中，对于并发运行上百个进程的系统，造成极大的主存资源浪费
- 库函数（如printf）被合并可在执行目标中，磁盘上存放着数千个可执行文件，造成磁盘空间的极大浪费
- 程序员需关注是否有函数库的新版本出现，并须定期下载、重新编译和链接，更新困难、使用不便



# 动态链接的共享库

- 解决方案: Shared Libraries (共享库)
  - 是一个目标文件, 包含有代码和数据
  - 从程序中分离出来, 磁盘和内存中都只有一个备份
  - 可以动态地在装入时或运行时被加载并链接
  - Window称其为动态链接库 (Dynamic Link Libraries, .dll文件)
  - Linux称其为动态共享对象 (Dynamic Shared Objects, .so文件)

# 共享库 (Shared Libraries)

- 在内存中只有一个备份，被所有进程共享，节省内存空间
- 一个共享库目标文件被所有程序共享链接，节省磁盘空间
- 共享库升级时，被自动加载到内存和程序动态链接，使用方便
- 共享库可分模块、独立、用不同编程语言进行开发，效率高
- 第三方开发的共享库可作为程序插件，使程序功能易于扩展

# 共享库 (Shared Libraries)

- 动态链接可以按以下两种方式进行：
- 在**第一次加载并运行时进行** (load-time linking).
  - Linux通常由动态链接器(ld-linux.so)自动处理
  - 标准C库 (libc.so) 通常按这种方式动态被链接
- 在**已经开始运行后进行**(run-time linking).
  - 在Linux中，通过调用 dlopen()等接口来实现
    - 分发软件包、构建高性能Web服务器等

# 加载时动态链接

- 程序头表中有一个特殊的段：INTERP
- 其中记录了动态链接器目录及文件名ld-linux.so

Program Headers:

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

# 自定义一个动态共享库文件

myproc1.c

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

myproc2.c

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

PIC: Position Independent Code

位置无关代码

- 1) 保证共享库代码的位置可以是不确定的
- 2) 即使共享库代码的长度发生变化, 也不会影响调用它的程序

gcc -c myproc1.c myproc2.c

位置无关的共享代码库文件

gcc -shared -fPIC -o mylib.so myproc1.o myproc2.o

# 加载时动态链接

gcc -c main.c

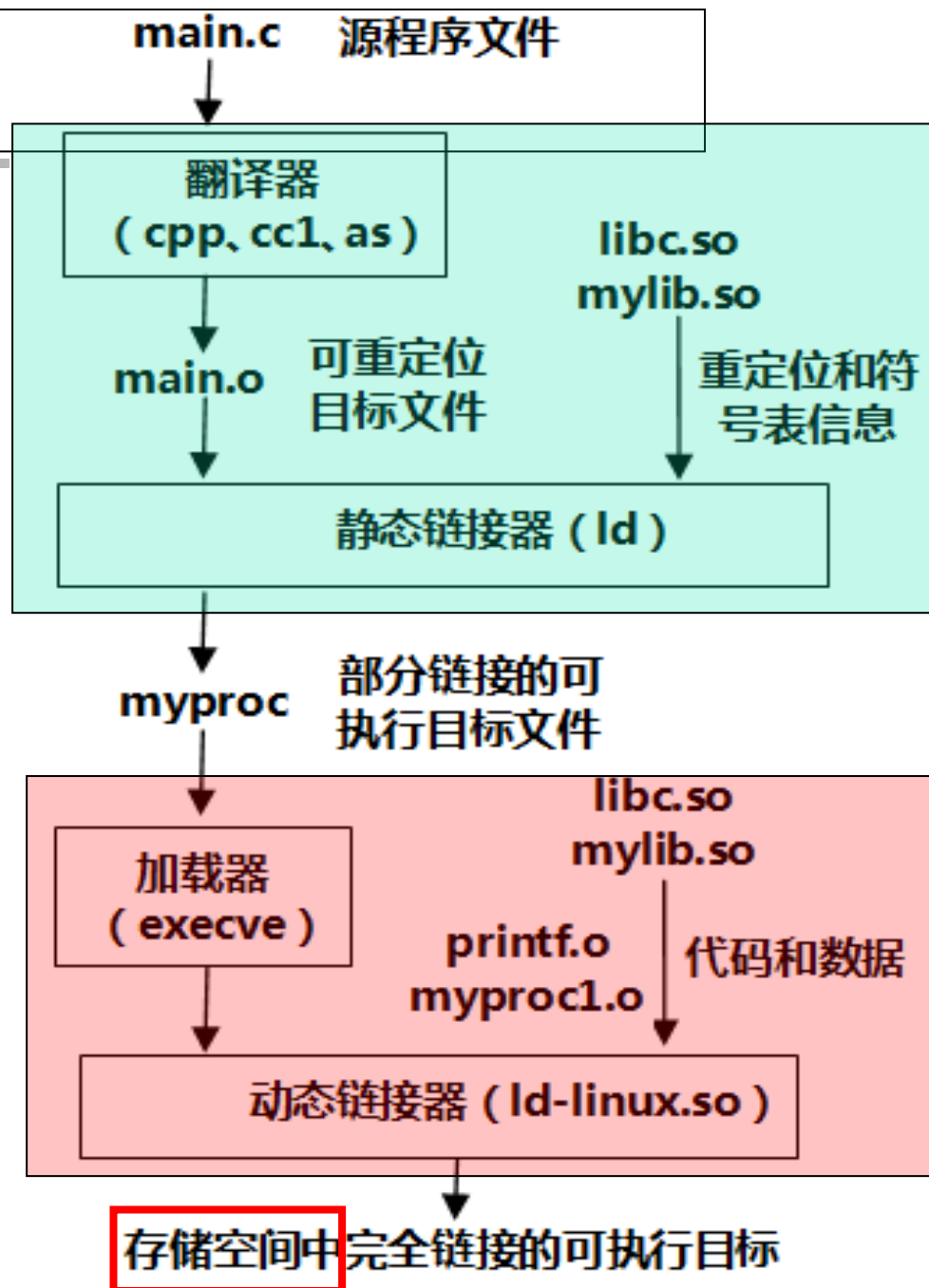
gcc -o myproc main.o **./mylib.so**

- 部分链接

- 只链接符号，无值

- 完全链接

- 确实地址



# 加载时动态链接

- 当加载和运行myproc（部分链接）时
  - Myproc内包含.interp节，其中包含了动态链接器的路径
  - 加载和运行动态链接器
  - 动态链接器的工作：
    - 重定位libc.so的代码和数据到某个内存段
    - 重定位mylib.so的代码和数据到某个内存段
    - 重定位myproc中所有对libc.so和mylib.so定义的符号与引用

# 运行时动态链接

- 可通过动态链接器接口提供的函数在运行时进行动态链接
- 类UNIX系统中的动态链接器接口定义了相应的函数，如 dlopen, dlsym, dlerror, dlclose等，其头文件为 dlfcn.h

```
#include <stdio.h>
#include <dlfcn.h>
int main()
{
    void *handle;
    void (*myfunc1)();
    char *error;
    /* 动态装入包含函数myfunc1()的共享库文件 */
    handle = dlopen("./mylib.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    /* 获得一个指向函数myfunc1()的指针myfunc1 */
    myfunc1 = dlsym(handle, "myfunc1");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }
    /* 现在可以像调用其他函数一样调用函数myfunc1() */
    myfunc1();
    /* 关闭（卸载）共享库文件 */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```



# 位置无关代码 (PIC)

- 动态链接用到一个重要概念：
  - 位置无关代码 (Position-Independent Code, PIC)
  - GCC选项-fPIC指示生成PIC代码
- 共享库代码是一种PIC
  - 共享库代码的位置可以是不确定的
  - 即使共享库代码的长度发生变化, 也不影响调用它的程序
- 引入PIC的目的
  - 链接器无需修改代码即可将共享库加载到任意地址运行
- 所有引用情况
  - (1) 模块内的过程调用、跳转, 采用PC相对偏移寻址
  - (2) 模块内数据访问, 如模块内的全局变量和静态变量
  - (3) 模块外的过程调用、跳转
  - (4) 模块外的数据访问, 如外部变量的访问

要生成PIC代码, 主要解决这两个问题

# (1) 模块内部函数调用或跳转

- 调用或跳转源与目的地都在同一个模块，相对位置固定，只要用相对偏移寻址即可
- 无需动态链接器进行重定位

8048344 <bar>:

```
8048344: 55          pushl %ebp
8048345: 89 e5       movl %esp, %ebp
.....
8048352: c3         ret
8048353: 90         nop
```

8048354 <foo>:

```
8048354: 55          pushl %ebp
.....
8048364: e8 db ff ff ff  call 8048344 <bar>
8048369:
.....
```

```
static int a;
static int b;
extern void ext();
```

```
void bar()
```

```
{
    a=1;
    b=2;
}
```

```
void foo()
```

```
{
    bar();
    ext();
}
```

**call的目标地址为:**

**0x8048369+**  
**0xffffffffdb(-0x25)=**  
**0x8048344**

**JMP指令也可用相**  
**对寻址方式解决**

## (2) 模块内部数据引用

- .data节与.text节之间的相对位置确定，任何引用局部符号的指令与该符号之间的距离是一个常数

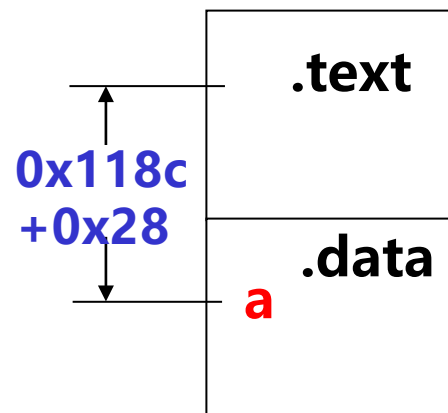
```
0000344 <bar>:
0000344: 55                pushl %ebp
0000345: 89 e5             movl  %esp, %ebp
0000347: e8 50 00 00 00    call 39c <__get_pc>
000034c: 81 c1 8c 11 00 00 addl  $0x118c, %ecx
0000352: c7 81 28 00 00 00 movl  $0x1, 0x28(%ecx)
.....
0000362: c3                ret

000039c <__get_pc>:
000039c: 8b 0c 24          movl  (%esp), %ecx
000039f: c3                ret
```

```
static int a;
extern int b;
extern void ext();

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

多用了4条指令



变量a与引用a的指令之间的距离为常数，调用\_\_get\_pc后，call指令的返回地址被置ECX。若模块被加载到0x9000000，则a的访问地址为：

**0x9000000 + 0x34c + 0x118c(指令与.data间距离) + 0x28(a在.data节中偏移)**



## (4) 模块间调用、跳转

- **方法一**：类似于(3)，在GOT中加一个项(指针)，用于指向目标函数的首地址（如&ext）
- **动态加载时**，填入目标函数的首地址

0000050c <foo>:

0000050c: 55

pushl %ebp

.....

00000557: e8 00 00 00 00 call 0000055c

0000055c: 5b

popl %ebx

0000055d:

addl \$1204, %ebx

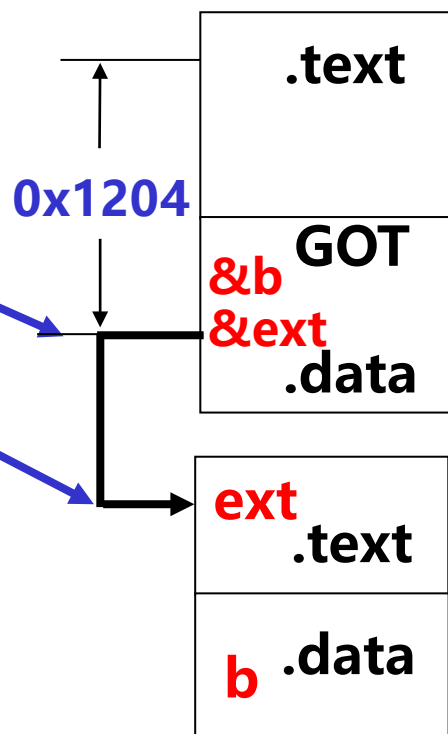
.....

call \*(%ebx)

.....

**\*(%ebx)为间接地址:  $R[eip] \leftarrow M[R[ebx]]$**

```
static int a;  
extern int b;  
extern void ext();  
  
void foo()  
{  
    bar();  
    ext();  
}  
.....
```



**启动时动态绑定GOT，降低启动时效率**

**共享库模块**

## (4) 模块间调用、跳转

### ■ 延迟绑定

#### ■ GOT

- GOT[0]: .dynamic节首址, 该节中包含动态链接器所需要的基本信息, 如符号表位置、重定位表位置等;
- GOT[1]: 动态链接器的标识信息
- GOT[2]: 动态链接器延迟绑定代码的入口地址
- 从GOT[3]开始, 对应调用的共享库函数

#### ■ PLT

- PLT是.text节一部分, 结构数组, 每项16B
- 除PLT[0]外, 其余项各对应一个共享库函数
- 一组指令

## (4) 模块间调用、跳转

- 延迟绑定
  - 初次调用

### PLT[0]

```
0804833c: ff 35 88 95 04 08  pushl 0x8049588
8048342: ff 25 8c 95 04 08  jmp  *0x804958c
8048348: 00 00 00 00
```

### PLT[1] <ext>      用 ID=0 标识ext()函数

```
0804834c: ff 25 90 95 04 08  jmp  *0x8049590
8048352: 68 00 00 00 00    pushl $0x0
8048357: e9 e0 ff ff      jmp  804833c
```

804833c	:	PLT[0]
804834c	:	PLT[1]
	.text	
8049584	0804956c	GOT[0]
8049588	4000a9f8	GOT[1]
804958c	4000596f	GOT[2]
8049590	08048352	GOT[3]
	.data	

可执行文件foo

## (4) 模块间调用、跳转

- 延迟绑定
  - 初次调用

ext()的调用指令:

804845b: e8 ec fe ff ff **call 804834c <ext>**

**PLT[0]**

```
0804833c: ff 35 88 95 04 08  pushl 0x8049588
8048342: ff 25 8c 95 04 08  jmp  *0x804958c
8048348: 00 00 00 00
```

**PLT[1] <ext>**      用 ID=0 标识ext()函数

```
0804834c: ff 25 90 95 04 08  jmp  *0x8049590
8048352: 68 00 00 00 00    pushl $0x0
8048357: e9 e0 ff ff ff    jmp  804833c
```

804833c	:	PLT[0]
804834c	:	PLT[1]
	.text	
8049584	0804956c	GOT[0]
8049588	4000a9f8	GOT[1]
804958c	4000596f	GOT[2]
8049590	08048352	GOT[3]
	.data	

可执行文件foo



## (4) 模块间调用、跳转

- 延迟绑定
  - 初次调用

ext()的调用指令:

804845b: e8 ec fe ff ff **call 804834c <ext>**

**PLT[0]**

```
0804833c: ff 35 88 95 04 08  pushl 0x8049588
8048342: ff 25 8c 95 04 08  jmp  *0x804958c
8048348: 00 00 00 00
```

**PLT[1] <ext>**      用 ID=0 标识ext()函数

```
0804834c: ff 25 90 95 04 08  jmp *0x8049590
8048352: 68 00 00 00 00    pushl $0x0
8048357: e9 e0 ff ff ff    jmp  804833c
```

804833c	:	PLT[0]
804834c	:	PLT[1]
	.text	
8049584	0804956c	GOT[0]
8049588	4000a9f8	GOT[1]
804958c	4000596f	GOT[2]
8049590	08048352	GOT[3]
	.data	

可执行文件foo

## (4) 模块间调用、跳转

### ■ 延迟绑定

#### ■ 初次调用完成之后

ext()的调用指令:

804845b: e8 ec fe ff ff **call 804834c <ext>**

#### PLT[0]

```
0804833c: ff 35 88 95 04 08  pushl 0x8049588
8048342: ff 25 8c 95 04 08  jmp  *0x804958c
8048348: 00 00 00 00
```

#### PLT[1] <ext>      用 ID=0 标识ext()函数

```
0804834c: ff 25 90 95 04 08  jmp *0x8049590
8048352: 68 00 00 00 00    pushl $0x0
8048357: e9 e0 ff ff ff    jmp  804833c
```

804833c	⋮	PLT[0]
804834c	⋮	PLT[1]
	.text	
8049584	0804956c	GOT[0]
8049588	4000a9f8	GOT[1]
804958c	4000596f	GOT[2]
8049590	7fffxxx	GOT[3]
	.data	

可执行文件foo

# 课后作业

- 以课堂讲授的代码为例，利用不同的参数，按步骤拆分gcc的编译过程
  - 预处理、编译、汇编、链接
  - 利用readelf/objdump等工具查看并分析目标文件和可执行文件的格式
  - 观察main.o和swap.o里面的重定位项
  - 观察并记录重定位条目在可执行文件中的变化

```
int buf[2] = {1, 2};  
void swap();
```

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

**main.c**

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

```
void swap()  
{
```

**swap.c**

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

```
}
```

# 课后作业

- 以下列代码为例，生成静态库和动态库
- 对比静态链接和动态链接所得的可执行文件
  - 观察文件大小、结构等不同点

**main.c**

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

**myproc1.c**

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

**myproc2.c**

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

# 课后作业

- 以下列代码为对象，采用动态链接生成可执行文件
- 利用gdb调试并分析程序动态链接过程中的GOT/PLT机制
- 期末交作业

main.c

```
int main()
{
    printf( "hello world\n" );
    return 0;
}
```