Chapter4 程序的链接

第一讲 目标文件格式

使用链接的好处

模块化

（1）一个程序可以分成很多源程序文件

（2）可构建公共函数库，如数学库，标准C库等

效率高

（1）时间上，可分开编译

只需重新编译被修改的源程序文件，然后重新链接

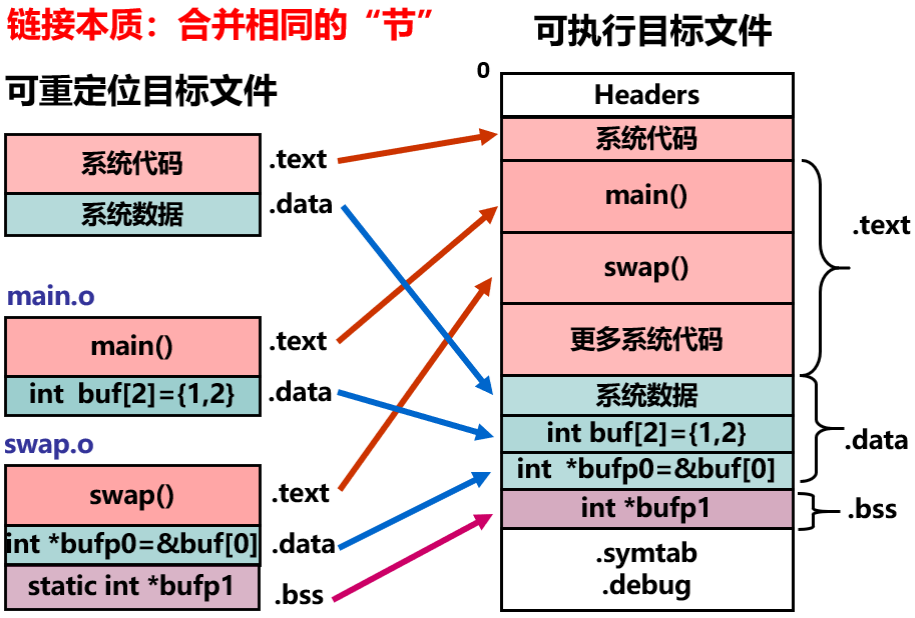
（2）空间上，无需包含共享库所有代码

源文件中无需包含共享库函数的源码，只要直接调用即可

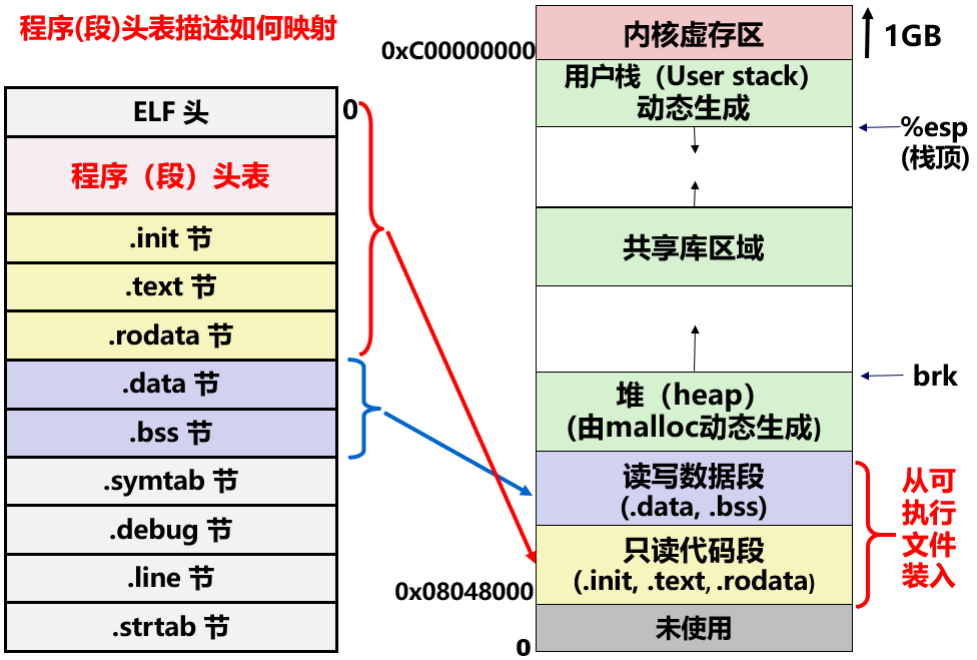
可执行文件和运行时的内存中只需包含所调用函数的代码

而不需要包含整个共享库

链接过程的本质



可执行文件的存储器映像



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

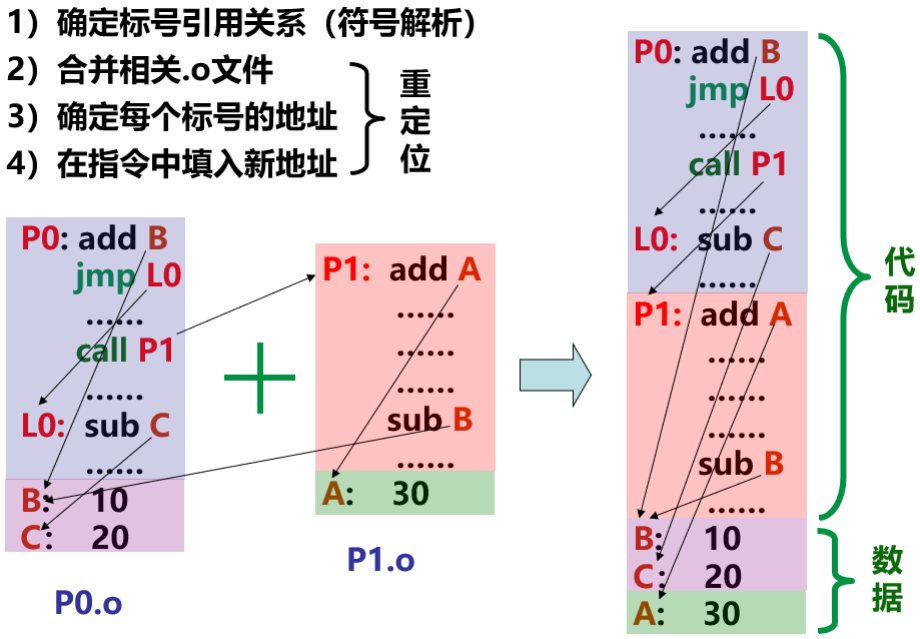
链接器将每个符号的引用都与一个确定的符号定义建立关联

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



三类目标文件

目标代码（Object Code）指编译器和汇编器处理源代码后所生成的机器语言目标代码

目标文件（Object File）指包含目标代码的文件

- **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)

Executable and Linkable Format (ELF)

- **两种视图**
  - **链接视图（被链接）：Relocatable object files**
  - **执行视图（被执行）：Executable object files**

链接视图

节（section）是ELF 文件中具有相同特征的最小可处理单位
.text节: 代码
.data节: 数据
.rodata: 只读数据
.bss: 未初始化数据

执行视图

由不同的段（segment）组成，描述节如何映射到存储段中，可多个节映射到同一段，如：可合并.data节和.bss节,并映射到一个可读可写数据段中

链接视图—可重定位目标文件

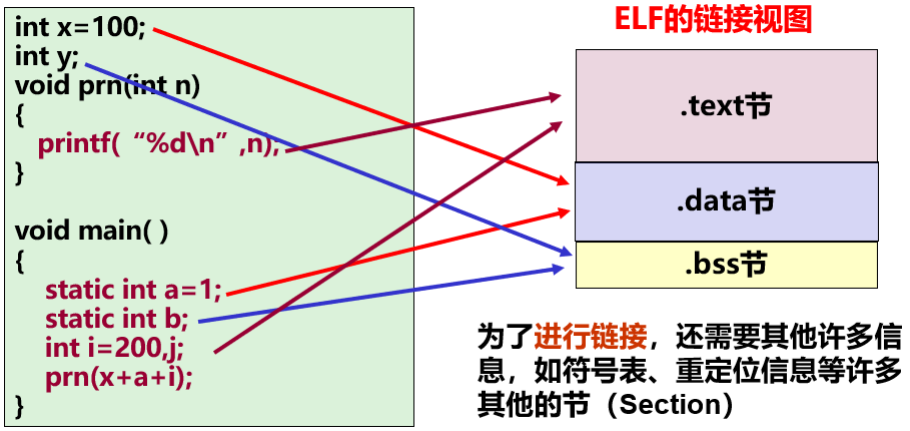
可被链接（合并）生成可执行文件或共享目标文件

静态链接库文件由若干个可重定位目标文件组成

包含代码、数据（已初始化.data和未初始化.bss）

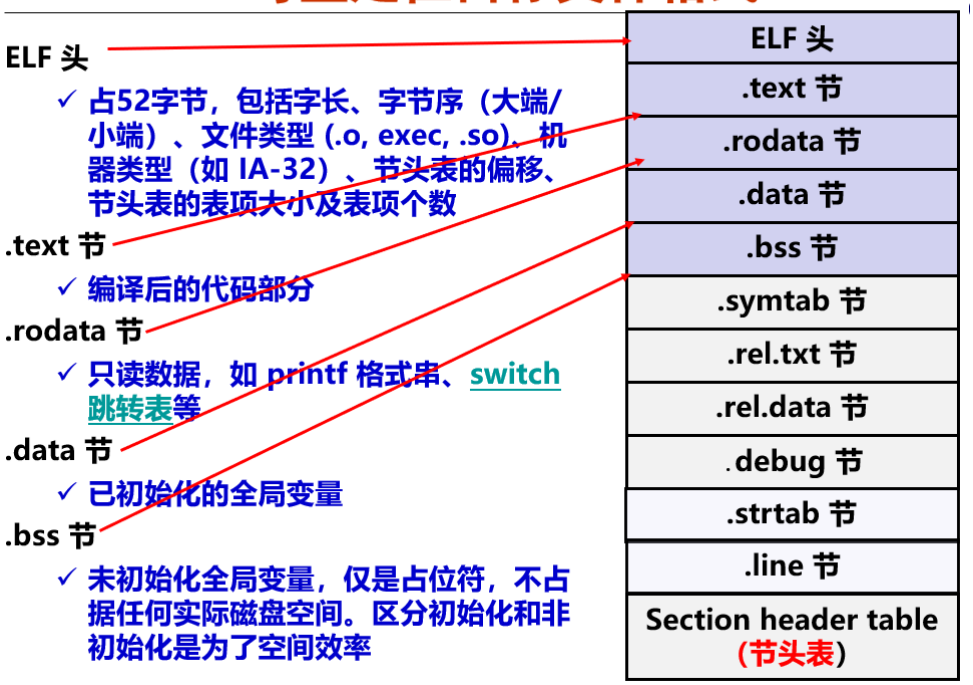
包含重定位信息（指出哪些符号引用处需要重定位）

文件扩展名为.o（相当于Windows中的 .obj文件）



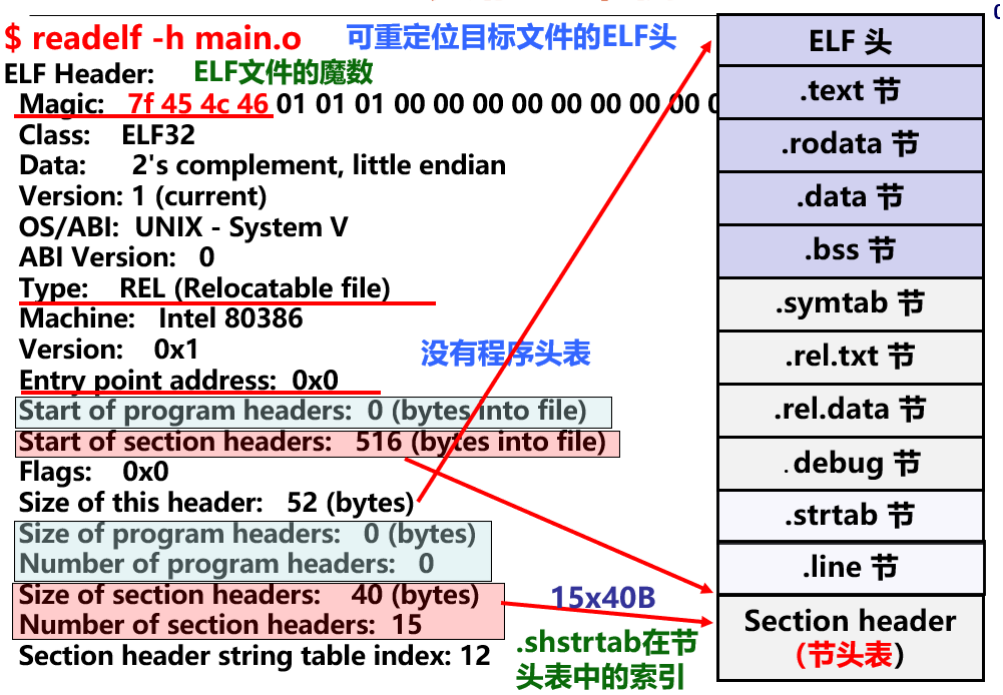
为了进行链接，还需要其他许多信息，如符号表、重定位信息等许多其他的节（Section）

可重定位目标文件格式



- ELF头（ELF Header）

ELF头信息举例



- 节头表（Section Header Table）

- 除ELF头之外，节头表是ELF可重定位目标文件中最重要的部分内容
- 描述每个节的节名、在文件中的偏移、大小、访问属性、对齐方式等
- 以下是32位系统对应的数据结构（每个表项占40B）

```
typedef struct {
    Elf32_Word    sh_name;   节名字符串在.shstrtab中的偏移
    Elf32_Word    sh_type;   节类型：无效/代码或数据/符号/字符串/...
    Elf32_Word    sh_flags;  节标志：该节在虚拟空间中的访问属性
    Elf32_Addr    sh_addr;   虚拟地址：若可被加载，则对应虚拟地址
    Elf32_Off     sh_offset; 在文件中的偏移地址，对.bss节而言则无意义
    Elf32_Word    sh_size;   节在文件中所占的长度
    Elf32_Word    sh_link;   sh_link和sh_info用于与链接相关的节（如
    Elf32_Word    sh_info;   .rel.text节、.rel.data节、.symtab节等）
    Elf32_Word    sh_addralign; 节的对齐要求
    Elf32_Word    sh_entsize; 节中每个表项的长度，0表示无固定长度表项
} Elf32_Shdr;
```

节头表信息举例

$ readelf -S test.o

There are 11 section headers, starting at offset 0x120:

Section Headers:

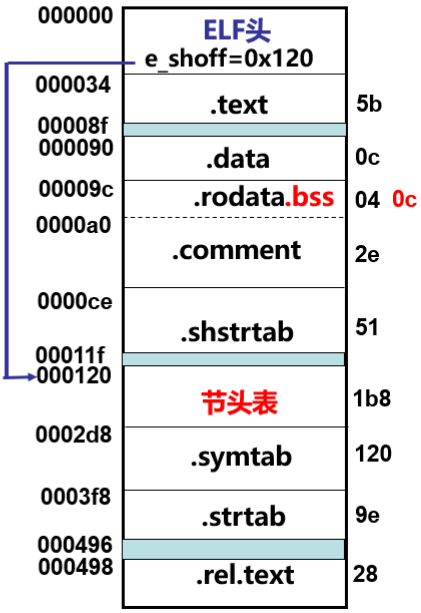
[Nr] Name	Off	Size	ES	Flg	Lk	Inf	Al
[0]	000000	000000	00		0	0	0
[1] .text	000034	00005b	00	AX	0	0	4
[2] .rel.text	000498	000028	08		9	1	4
[3] .data	000090	00000c	00	WA	0	0	4
[4] .bss	00009c	00000c	00	WA	0	0	4
[5] .rodata	00009c	000004	00	A	0	0	1
[6] .comment	0000a0	00002e	00		0	0	1
[7] .note.GNU-stack	0000ce	000000	00		0	0	1
[8] .shstrtab	0000ce	000051	00		0	0	1
[9] .symtab	0002d8	000120	10		10	13	4
[10] .strtab	0003f8	00009e	00		0	0	1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)

.....

有4个节将会分配（A）存储空间
.text：可执行
.data和.bss：可读可写
.rodata：可读

可重定位目标文件test.o的结构



执行视图—可执行目标文件

包含代码、数据（已初始化.data和未初始化.bss）

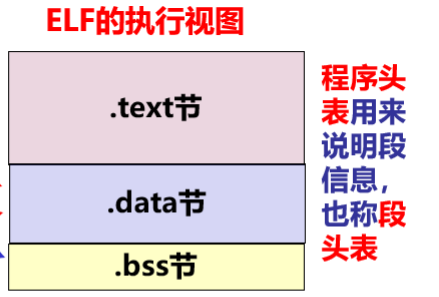
定义的所有变量和函数已有确定地址（虚拟地址空间中的地址）

符号引用处已被重定位，以指向所引用的定义符号

没有文件扩展名或默认为a.out（相当于Windows中的 .exe文件）

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

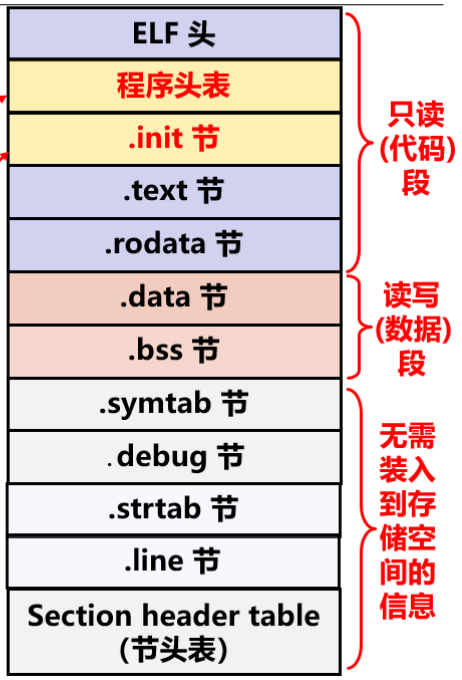
```
int x=100;
int y;
void prn(int n)
{
  printf( "%d\n" ,n);
}

void main( )
{
   static int a=1;
   static int b;
   int i=200,j;
   prn(x+a+i);
}
```



为了能执行，还需将具相同访问属性的节合并成段（Segment），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

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（ELF Header）

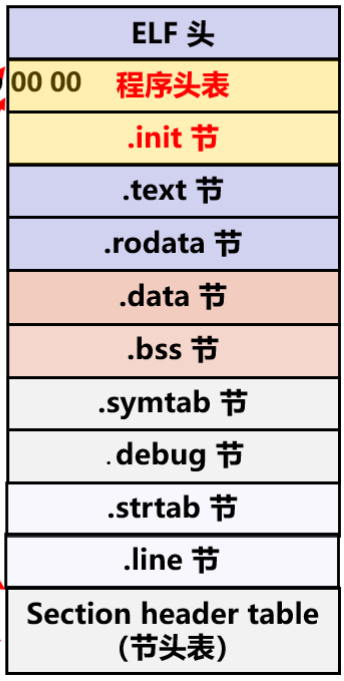
ELF头信息举例

```
$ 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
```

可执行目标文件的ELF头

8x32B

29x40B



- 程序头表（Program Header）

可执行文件中的程序头表

```
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;
```

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

以下是某可执行目标文件程序头表信息

有8个表项，其中两个为可装入段（即Type=LOAD）

$ readelf –l main

```
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
```

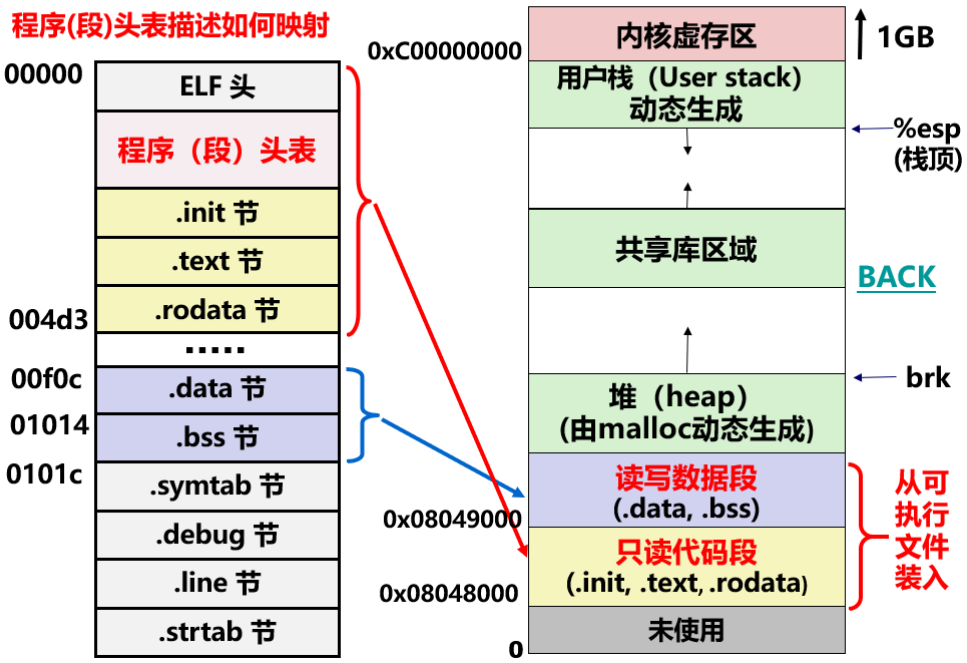
SKIP

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=2^{12}=4KB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.data节，映射到虚拟地址0x8049f0c开始长度为0x110字节的存储区域，在0x110=272B存储区中，前0x108=264B用.data节内容初始化，后面272-264=8B对应.bss节，初始化为0，按0x1000=4KB对齐，具有可读可写权限（Flg=RW），是可读写数据段。

- 可执行文件的存储器映像

可执行文件的存储器映像



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

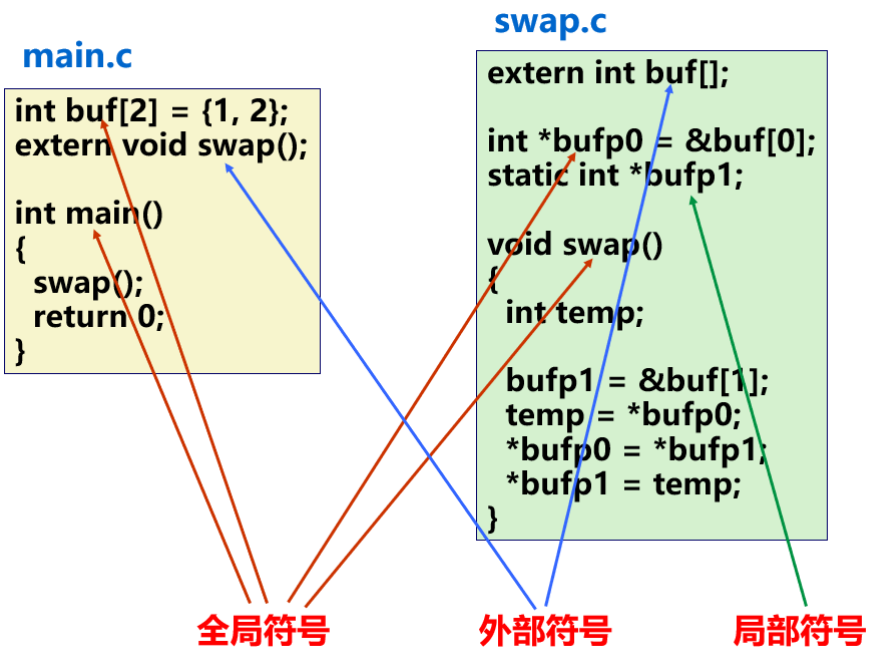
如，main.c 中的函数名swap

- **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

注：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



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

函数名在text节中

变量名在data节或bss节中

- 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*/
    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*/ 函数大小或变量长度
    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;
    short  section;  /*符号对应目标所在的节，或其他情况*/
} Elf_Symbol;
```

其他情况：ABS表示不该被重定位；UND表示未定义；COM表示未初始化数据（.bss），此时，value表示对齐要求，size给出最小大小

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。
- 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

```
add B
jmp L0
......
L0: sub 23
......
B:  ......
```

确定L0的地址，再在jmp指令中填入L0的地址

符号解析也称符号绑定

“符号的定义”其实质是什么？

指被分配了存储空间。为函数名即指其代码所在区；为变量名即指其所占的静态数据区。

所有定义符号的值就是其目标所在的首地址

1. 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

符号解析时只能有一个确定的定义（即每个符号仅占一处存储空间）

- 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。

2. 多重定义全局符号的问题

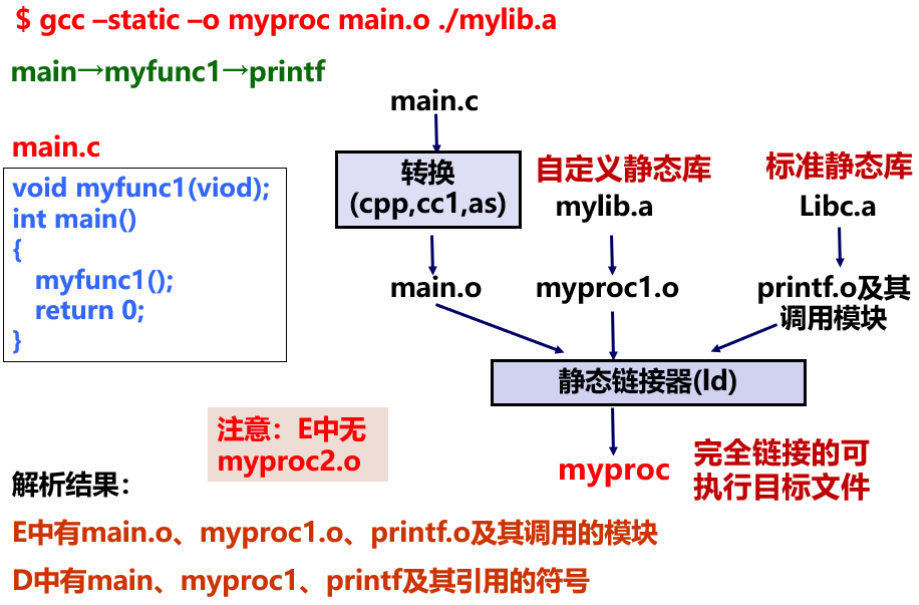
- **尽量避免使用全局变量**
- **一定需要用的话，就按以下规则使用**

  —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

3. 链接器中符号解析的全过程



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, ...)

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
```

main.c

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

```
$ gcc –c main.c
$ gcc –static –o myproc main.o ./mylib.a
```

libc.a无需明显指出!

链接器对外部引用的解析算法要点

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

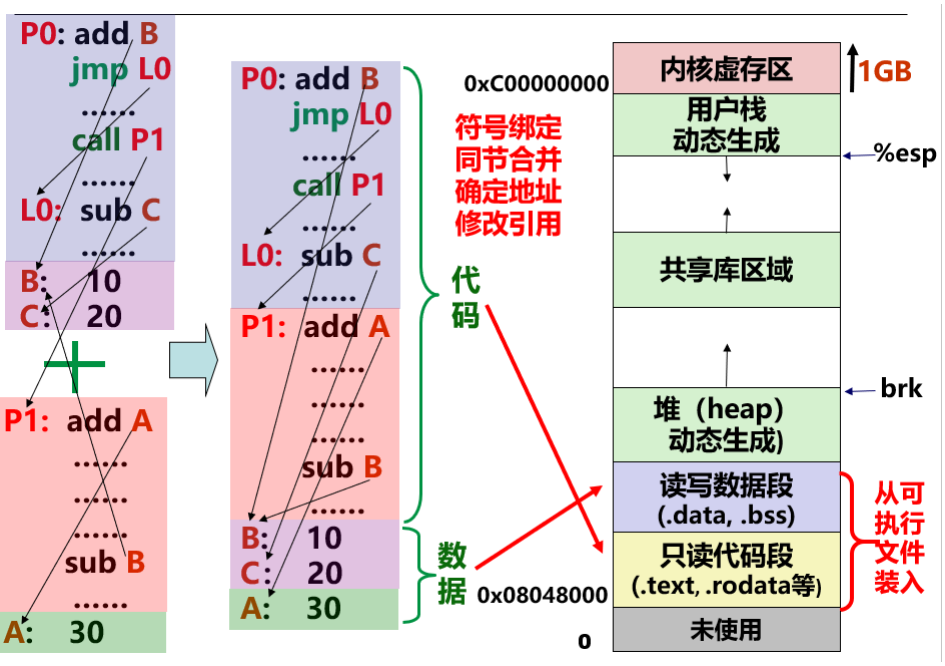
问题和对策

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

第三讲 重定位

链接操作的步骤



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

将集合E的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**
- **指令中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 ->全局变量**
  - **R_386_PC32: PC相对地址 ->函数调用**

例如，在rel_text节中有重定位条目如下

offset: 0x1	offset: 0x6
symbol: B	symbol: L0
type: R_386_32	type: R_386_PC32

问题：重定位条目和汇编后的机器代码在哪种目标文件中？

在可重定位目标（.o）文件中！

符号的定义：加上此处之后，符号表中会多一项

符号的引用：对该符号的使用，通常在数据初始化/代码段

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：

ADDR(r_sym) – ((ADDR(.text) + r_offset) – init)

引用目标处 **call指令下条指令地址** **即当前PC的值**

```
Disassembly of section .text:
00000000 <main>:
   ......
   6:    e8 fc ff ff ff      call   7 <main+0x7>
                值为-4     7: R_386_PC32 swap
   ......
```

- 转移目标地址=PC+偏移地址，PC=0x8048380+0x07-Init
- 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

- 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”

可执行目标文件中.data节的内容

```
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

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;
}
```

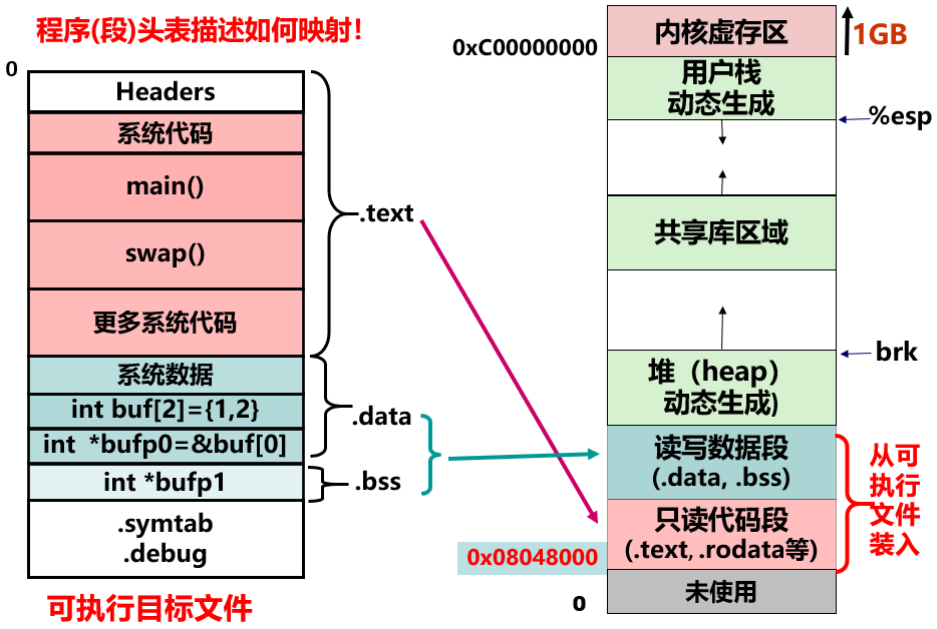
共有6处需要重定位

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
```

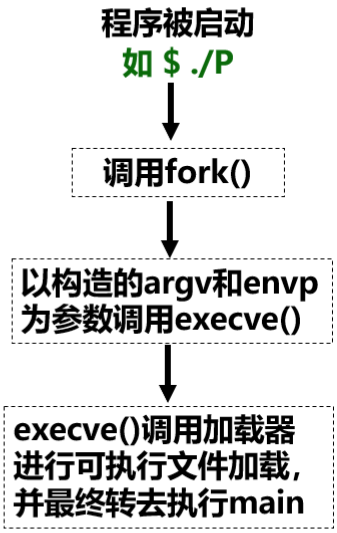
可执行文件的加载

可执行文件的存储器映像



可执行文件的加载

- 通过调用execve系统调用函数来调用加载器
- 加载器（loader）根据可执行文件的程序（段）头表中的信息，将可执行文件的代码和数据从磁盘“拷贝”到存储器中（实际上不会真正拷贝，仅建立一种映像，这涉及到许多复杂的过程和一些重要概念，将在后续课上学习）
- 加载后，将PC（EIP）设定指向Entry point (即符号_start处)，最终执行main函数，以启动程序执行。



第四讲 动态链接

静态库的缺点

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）被合并在可执行目标中，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

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）

动态链接的进行方式

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

动态链接的优点

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**

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
gcc –shared –fPIC –o mylib.so myproc1.o myproc2.o
```

位置无关的共享代码库文件

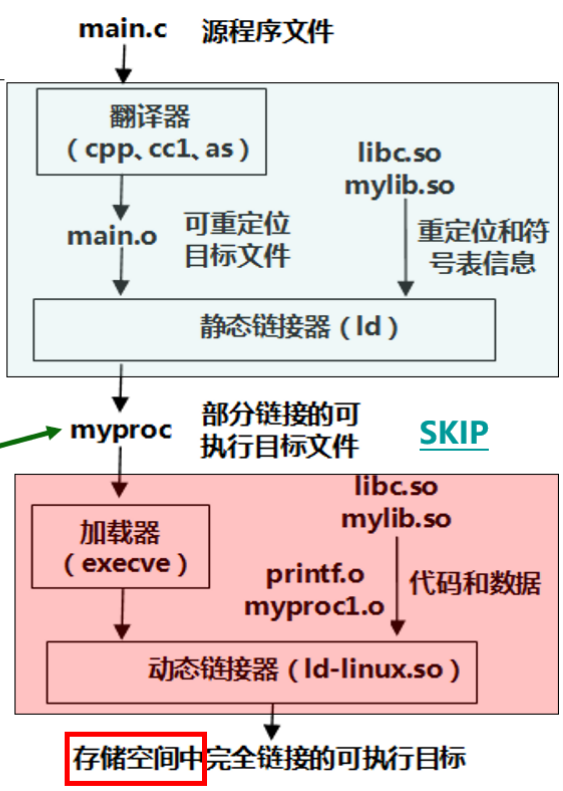
加载时动态链接

gcc –c main.c　libc.so无需明显指出
gcc –o myproc main.o ./mylib.so

调用关系：main→myfunc1→printf

main.c

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



加载 myproc 时，加载器发现在其程序头表中有 .interp 段，其中包含了动态链接器路径名 ld-linux.so，因而加载器根据指定路径加载并启动动态链接器运行。动态链接器完成相应的重定位工作后，再把控制权交给myproc，启动其第一条指令执行。

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

要实现动态链接，必须生成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

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

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

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

```
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:
 ......
```

call的目标地址为：
0x8048369+
0xffffffdb(-0x25)=
0x8048344

JMP指令也可用相对寻址方式解决

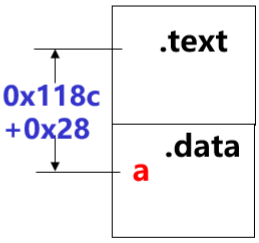
(2) 模块内部数据引用

- .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

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

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

```
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
```

多用了4条指令



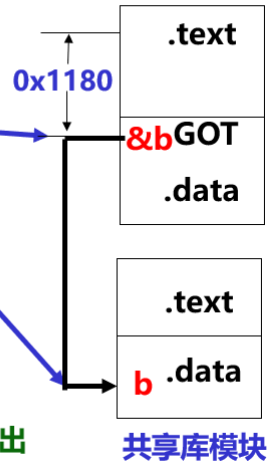
变量a与引用a的指令之间的距离为常数，调用__get_pc后，call指令的返回地址被置ECX。若模块被加载到0x9000000，则a的访问地址为：
0x9000000+0x34c+0x118c(指令与.data间距离)+0x28(a在.data节中偏移)

(3) 模块外数据的引用

- 引用其他模块的全局变量，无法确定相对距离
- 在.data节开始处设置一个指针数组（全局偏移表，GOT），指针可指向一个全局变量
- GOT与引用数据的指令之间相对距离固定

```
00000344 <bar>:
 00000344:   55                  pushl  %ebp
 ......
 00000357:   e8 00 00 00 00      call   0000035c
 0000035c:   5b                  popl   %ebx
 000035d:                        addl   $1180, %ebx
 ......                          movl   (%ebx), %eax
 ......                          movl   $2, (%eax)
 ......
```

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

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



- 编译器为GOT每一项生成一个重定位项（如.rel节...）
- 加载时，动态链接器对GOT中各项进行重定位，填入所引用的地址（如&b）

PIC有两个缺陷：多用4条指令；多了GOT（Global Offset Table），故需多用一个寄存器（如EBX），易造成寄存器溢出

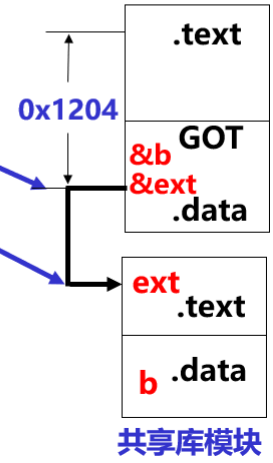
(4) 模块间调用、跳转

- 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]←M[R[ebx]]

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

void foo()
{
    bar();
    ext();
}
......
```



- 多用三条指令并额外多用一个寄存器（如EBX）

可用“延迟绑定（lazy binding）”技术减少指令条数：不在加载时重定位，而延迟到第一次函数调用时，需要用GOT和PLT（Procedure linkage Table, 过程链接表）

延迟绑定

(4) 模块间调用、跳转

```
extern  void ext();
void foo() {
    bar();
    ext();
}
……
```

方法二：延迟绑定

GOT是.data节一部分，开始三项固定，含义如下：

GOT[0]为.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]为动态链接器的标识信息

GOT[2]为动态链接器延迟绑定代码的入口地址

调用的共享库函数都有GOT项，如GOT[3]对应ext

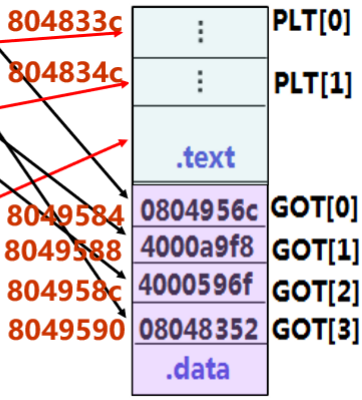
延时绑定代码根据GOT[1]和ID确定ext地址填入GOT[3]，并转ext执行，以后调用ext，只要多执行一条jmp指令而不是多3条指令。

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，如PLT[1]对应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
```

ext()的调用指令：

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



可执行文件foo

- 首次调用外部过程（例如addvec）时，过程对应的初始GOT表项指向相应PLT表项中pushl指令
  - 向栈中压入装载/重定位所需参数信息
  - 调用动态链接器（跳转至PLT[0]），装载/重定位相应模块
  - 动态链接器用实际的过程地址修改替换GOT[4]内容

- GOT

Address	Entry	Contents	Description
08049674	GOT[0]	0804969c	address of .dynamic section
08049678	GOT[1]	4000a9f8	identifying info for the linker
0804967c	GOT[2]	4000596f	entry point in dynamic linker
08049680	GOT[3]	0804845a	address of pushl in PLT[1] (printf)
08049684	GOT[4]	过程装载/重定位后的实际地址	

- PLT

```
PLT[0]
08048444:  ff 35 78 96 04 08  pushl  0x8049678   push &GOT[1]
 804844a:  ff 25 7c 96 04 08  jmp  *0x804967c    jmp to *GOT[2](linker)
 8048450:  00 00                                 padding
 8048452:  00 00                                 padding

PLT[1] <printf>
 8048454:  ff 25 80 96 04 08  jmp    *0x8049680  jmp to *GOT[3]
 804845a:  68 00 00 00 00     pushl  $0x0        ID for printf
 804845f:  e9 e0 ff ff ff     jmp    8048444     jmp to PLT[0]

PLT[2] <addvec>
过程调用
 8048464:  ff 25 84 96 04 08  jmp    *0x8049684  jump to *GOT[4]
 804846a:  68 08 00 00 00     pushl  $0x8        ID for addvec
 804846f:  e9 d0 ff ff ff     jmp    8048444     jmp to PLT[0]
```