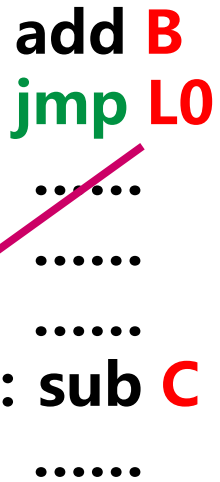
符号的重定位

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

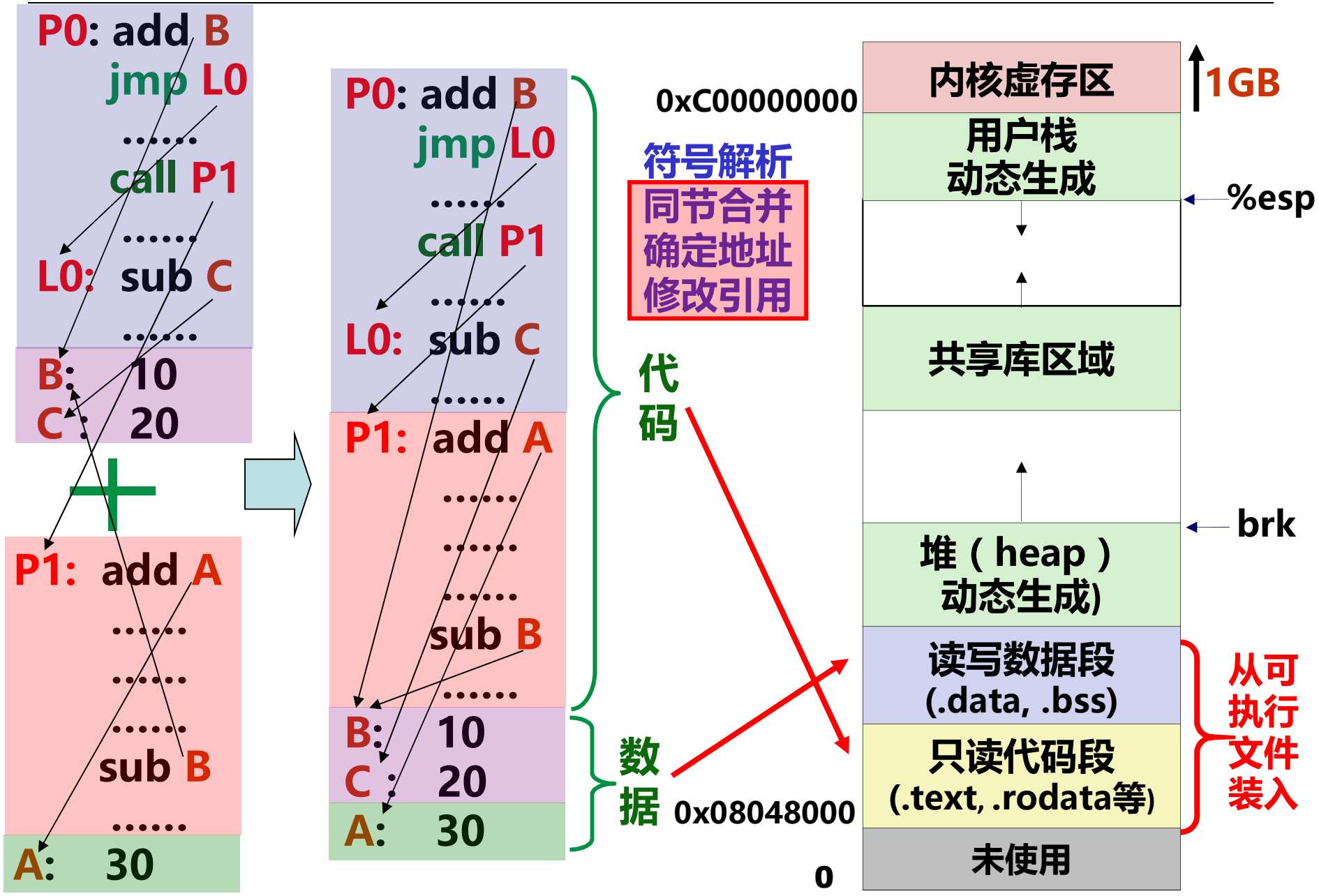
回顾：链接操作的步骤



- 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

回顾：链接操作的步骤


P0: add B
jmp L0
.....
call P1
.....
L0: sub C
.....
B: 10
C: 20
P1: add A
.....
.....
.....
sub B
.....
A: 30
P0: add B
jmp L0
.....
call P1
.....
L0: sub C
.....
P1: add A
.....
.....
.....
sub B
.....
B: 10
C: 20
A: 30
符号解析
同节合并
确定地址
修改引用
代码
数据
0xC0000000
0x08048000
0
内核虚存区
用户栈
动态生成
共享库区域
堆（heap）
动态生成)
读写数据段
(.data, .bss)
只读代码段
(.text, .rodata等)
未使用
1GB
%esp
brk
从可执行文件装入

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或局部变量都可确定地址
- 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）

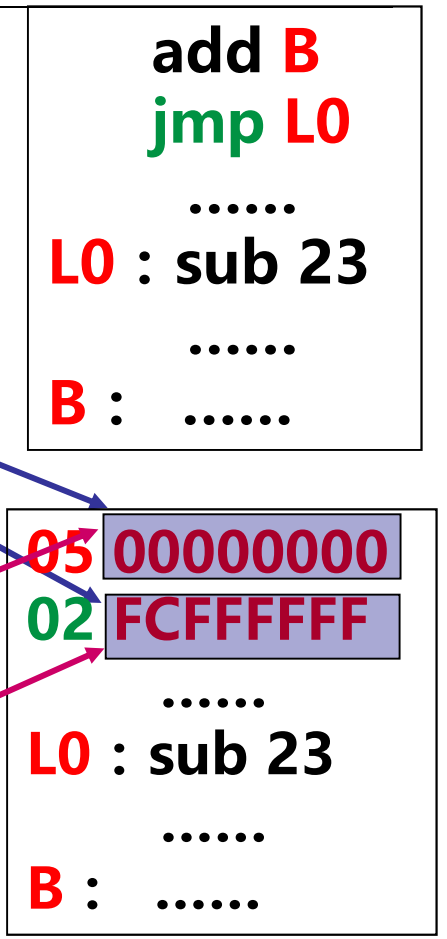
    需要用到在.rel_data和.rel_text节中保存的重定位信息

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
  - R_386_PC32: PC相对地址



例如，在rel_text节中有重定位条目

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

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！

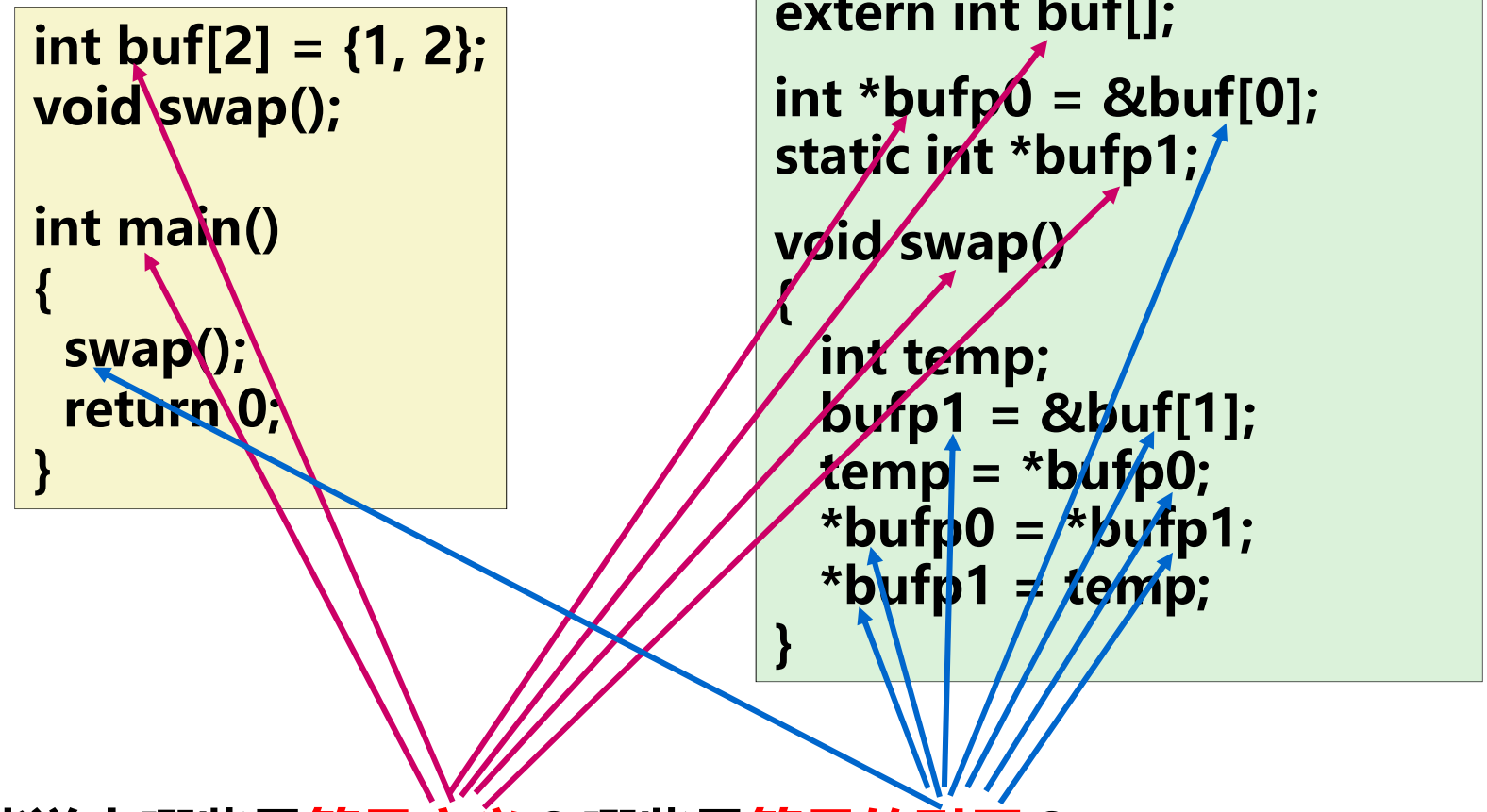
重定位操作举例

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;
}
```



你能说出哪些是符号定义？哪些是符号的引用？

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

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;
}
```

符号解析后的结果是什么？

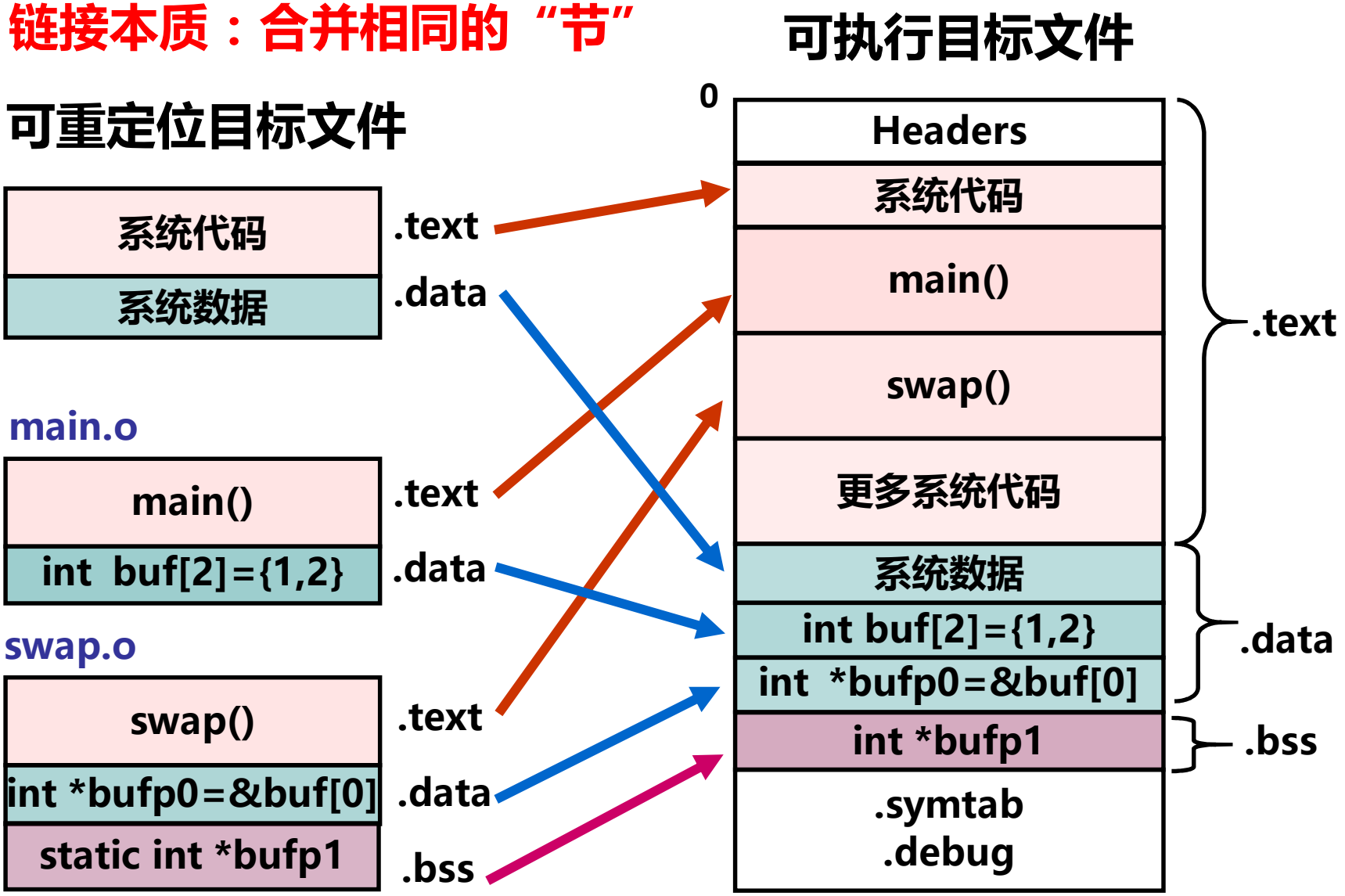
E中有main.o和swap.o两个模块！D中有所有定义的符号！

在main.o和swap.o的重定位条目中有重定位信息，反映符号引用的位置、绑定的定义符号名、重定位类型

用命令readelf -r main.o可显示main.o中的重定位条目（表项）

符号引用的地址需要重定位

链接本质：合并相同的“节”



main.o重定位前

main.c

```
int buf[2]={1,2};

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

main.o

```
Disassembly of section .text:
00000000 <main>:
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    83 e4 f0                and    $0xfffffff0,%esp
   6:    e8 fc ff ff ff          call   7 <main+0x7>
                                 7: R_386_PC32 swap
   b:    b8 00 00 00 00          mov    $0x0,%eax
  10:    c9                      leave
  11:    c3                      ret
```

main的定义在.text节中偏移为0处开始，占0x12B。

```
Disassembly of section .data:

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

buf的定义在.data节中偏移为0处开始，占8B。

SKIP

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。

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！

BACK

R_386_PC32的重定位方式

- 假定：
  - 可执行文件中mai
  - swap紧跟main后
- 则swap起始地址为
  - 0x8048380+0x12=0x8048392
  - 在4字节边界对齐的情况下，是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-init
  - PC=0x8048380+0x07-(-4)=0x804838b
  - 重定位值=转移目标地址-PC=0048394-0x804838b=0x9
  - call指令的机器代码为“e8 09 00 00 00”

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

重定位值

值为-4

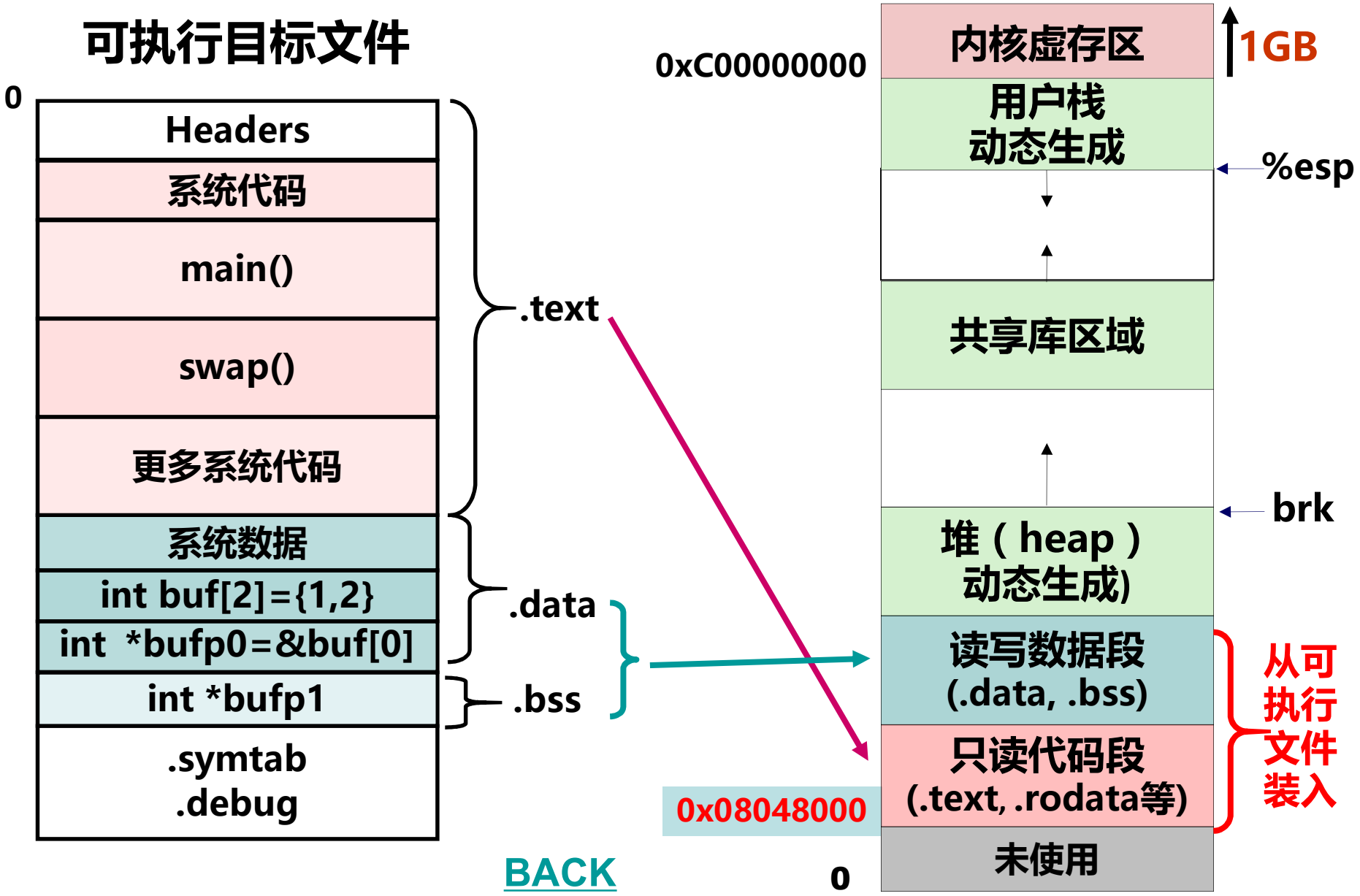
SKIP

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

确定定义符号的地址


可执行目标文件
0
Headers
系统代码
main()
swap()
更多系统代码
系统数据
int buf[2]={1,2}
int *bufp0=&buf[0]
int *bufp1
.symtab
.debug
.text
.data
.bss
0xC0000000
内核虚存区
1GB
用户栈
动态生成
%esp
共享库区域
brk
堆（heap）
动态生成)
读写数据段
(.data, .bss)
只读代码段
(.text, .rodata等)
从可执行文件装入
0x08048000
未使用
0
BACK

R_386_32的重定位方式

main.o中.data和.rel.data节内容

```
Disassembly of section .data:

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

buf定义在.data节中偏移为0处，占8B，没有需重定位的符号。

main.c

```
int buf[2]={1,2};

int main()
……
```

swap.c

```
extern int buf[];

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

void swap()
……
```

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

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，OBJDUMP工具解释后显示为“0 : R_386_32 buf”

r_sym=9说明引用的是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。

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，OBJDUMP工具解释后显示为“0：R_386_32 buf”

r_sym=9说明引用的是buf！

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
```

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 e4 f0        and   $0xfffffff0,%esp
 8048386:   e8 09 00 00 00  call  8048394 <swap>
 804838b:   b8 00 00 00 00  mov   $0x0,%eax
 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
```

你能写出该call指令的功能描述吗？

假定每个函数要求4字节边界对齐,故填充两条nop指令

R[eip]=0x804838b

1) R[esp]← R[esp]-4

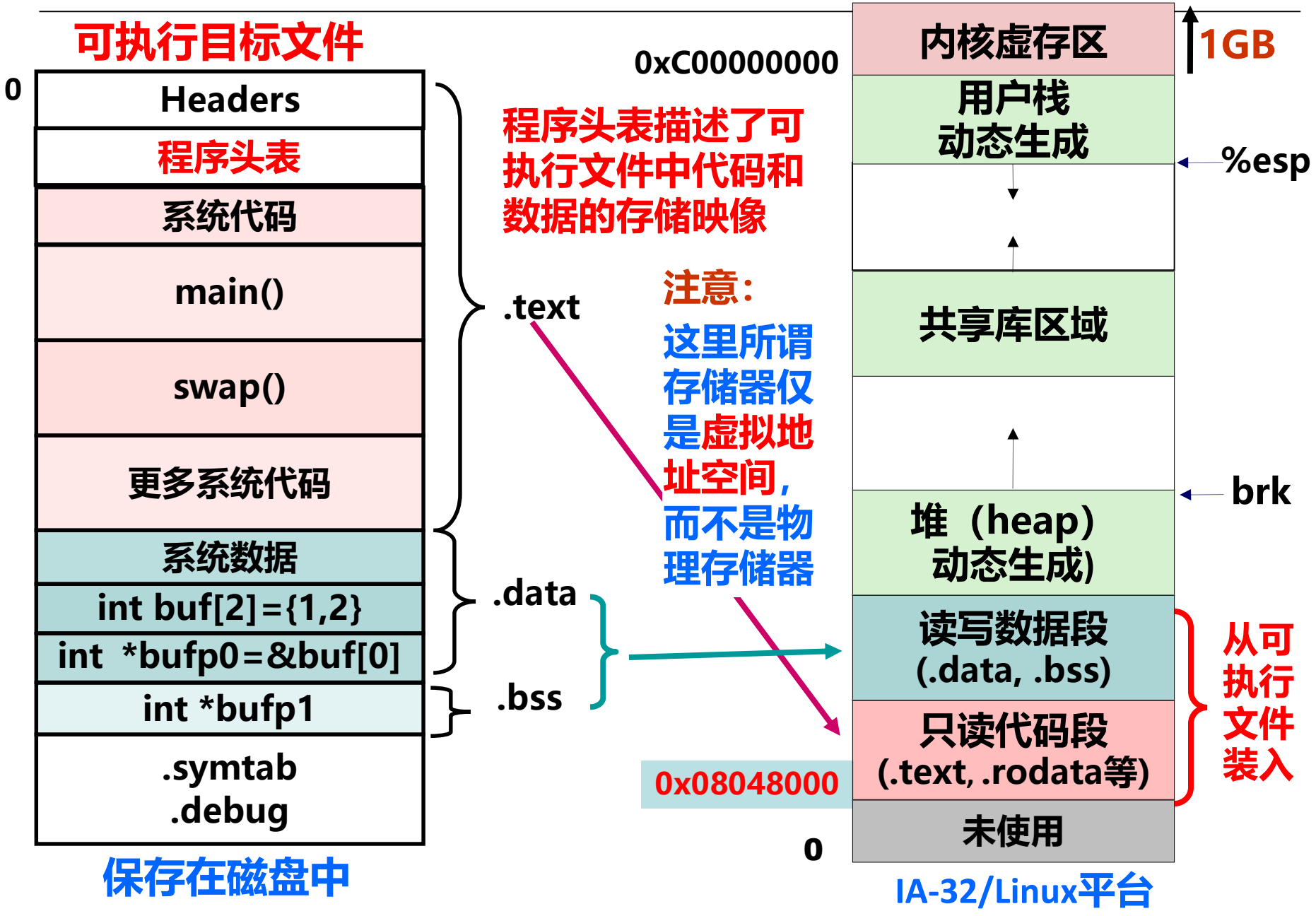
2) M[R[esp]] ←R[eip]

3) R[eip] ←R[eip]+0x9

可执行文件的加载

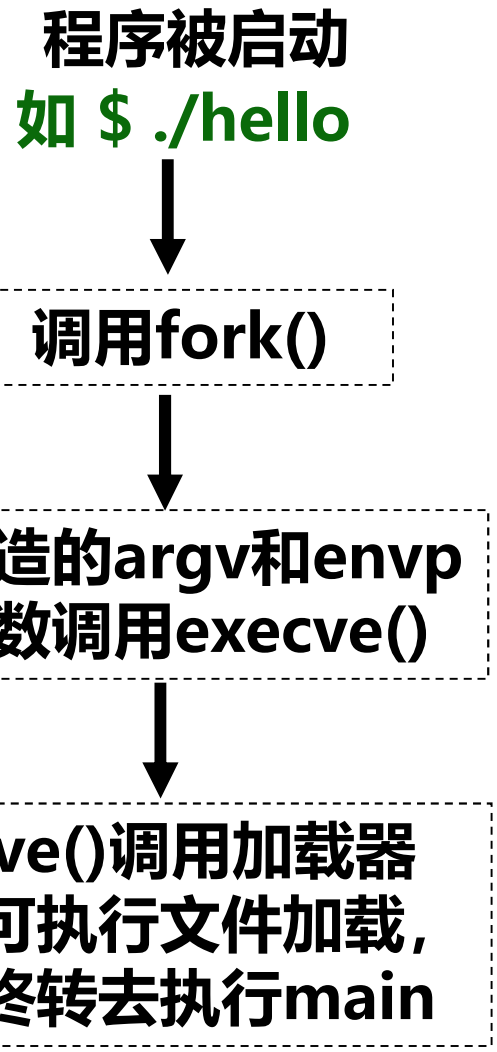
南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

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，仅建立一种映射，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



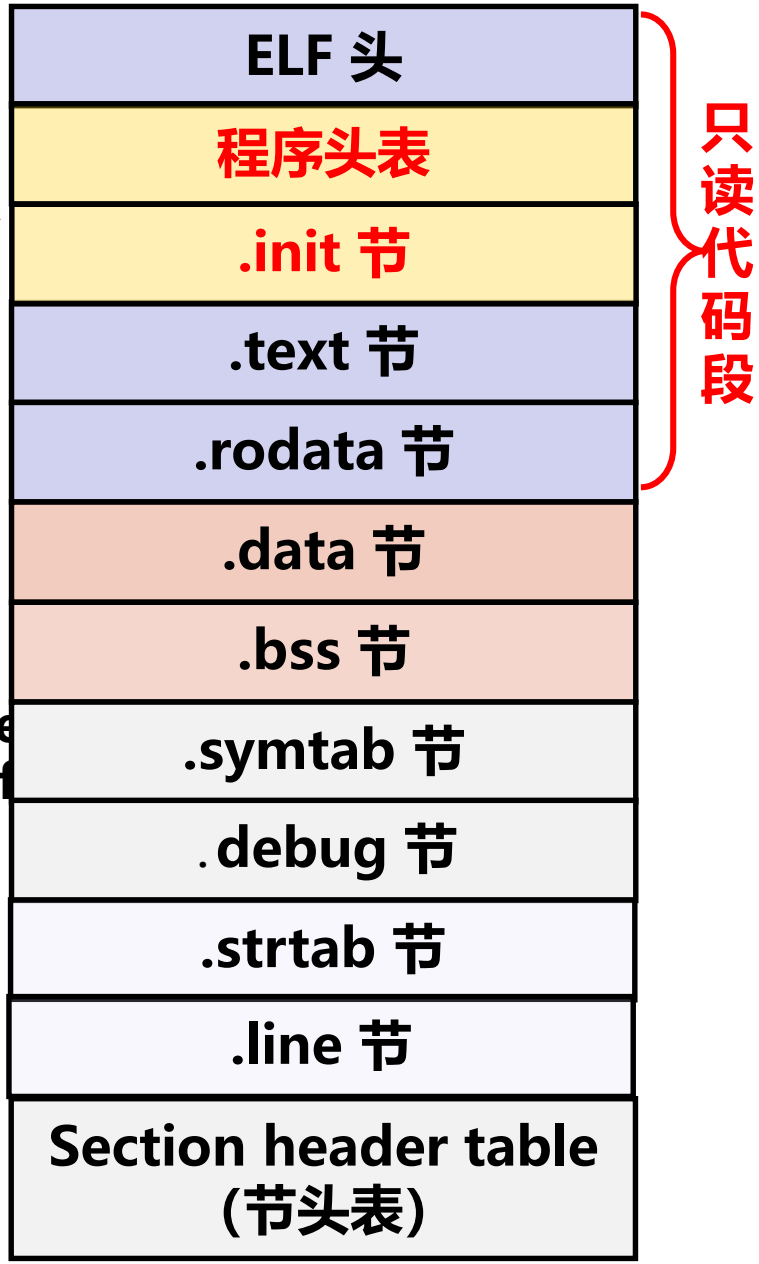
_start: __libc_init_first → _init → atexit → main → _exit

ELF头信息举例

BACK

```
$ readelf -h main
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 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 f
  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
```

注意：程序入口地址并不是0x8048000



程序的加载和运行

- UNIX/Linux系统中，可通过调用execve()函数来启动加载器。
- execve()函数的功能是在当前进程上下文中加载并运行一个新程序。execve()函数的用法如下：

  int execve(char *filename, char *argv[], *envp[]);

  filename是加载并运行的可执行文件名(如./hello)，可带参数列表argv和环境变量列表envp。若错误（如找不到指定文件filename），则返回-1，并将控制权交给调用程序； 若函数执行成功，则不返回，最终将控制权传递到可执行目标中的主函数main。
- 主函数main()的原型形式如下：

  int main(int argc, char **argv, char **envp); 或者：

  int main(int argc, char *argv[], char *envp[]);

  argc指定参数个数，参数列表中第一个总是命令名（可执行文件名）

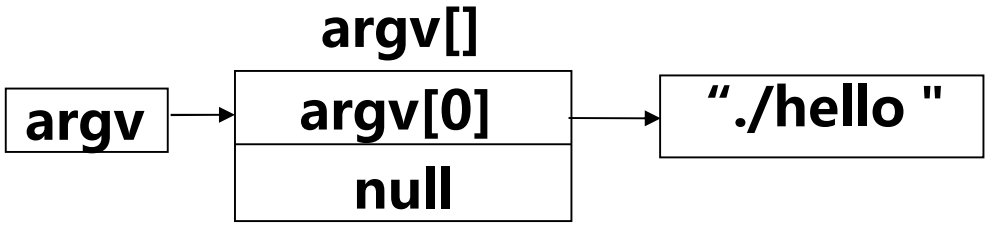
  例如：命令行为“ld -o test main.o test.o” 时，argc=5

程序的加载和运行

问题：hello程序的加载和运行过程是怎样的?

Step1：在shell命令行提示符后输入命令：$./hello[enter]

Step2：shell命令行解释器构造argv和envp



BACK

Step3：调用fork()函数，创建一个子进程，与父进程shell完全相同（只读/共享），包括只读代码段、可读写数据段、堆以及用户栈等。

Step4：调用execve()函数,在当前进程（新创建的子进程）的上下文中加载并运行hello程序。将hello中的.text节、.data节、.bss节等内容加载到当前进程的虚拟地址空间（仅修改当前进程上下文中关于存储映像的一些数据结构，不从磁盘拷贝代码、数据等内容）

Step5：调用hello程序的main()函数，hello程序开始在一个进程的上下文中运行。 int main(int argc, char *argv[], char *envp[]);

共享库和动态链接

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

动态链接的共享库（Shared Libraries）

- 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）

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

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

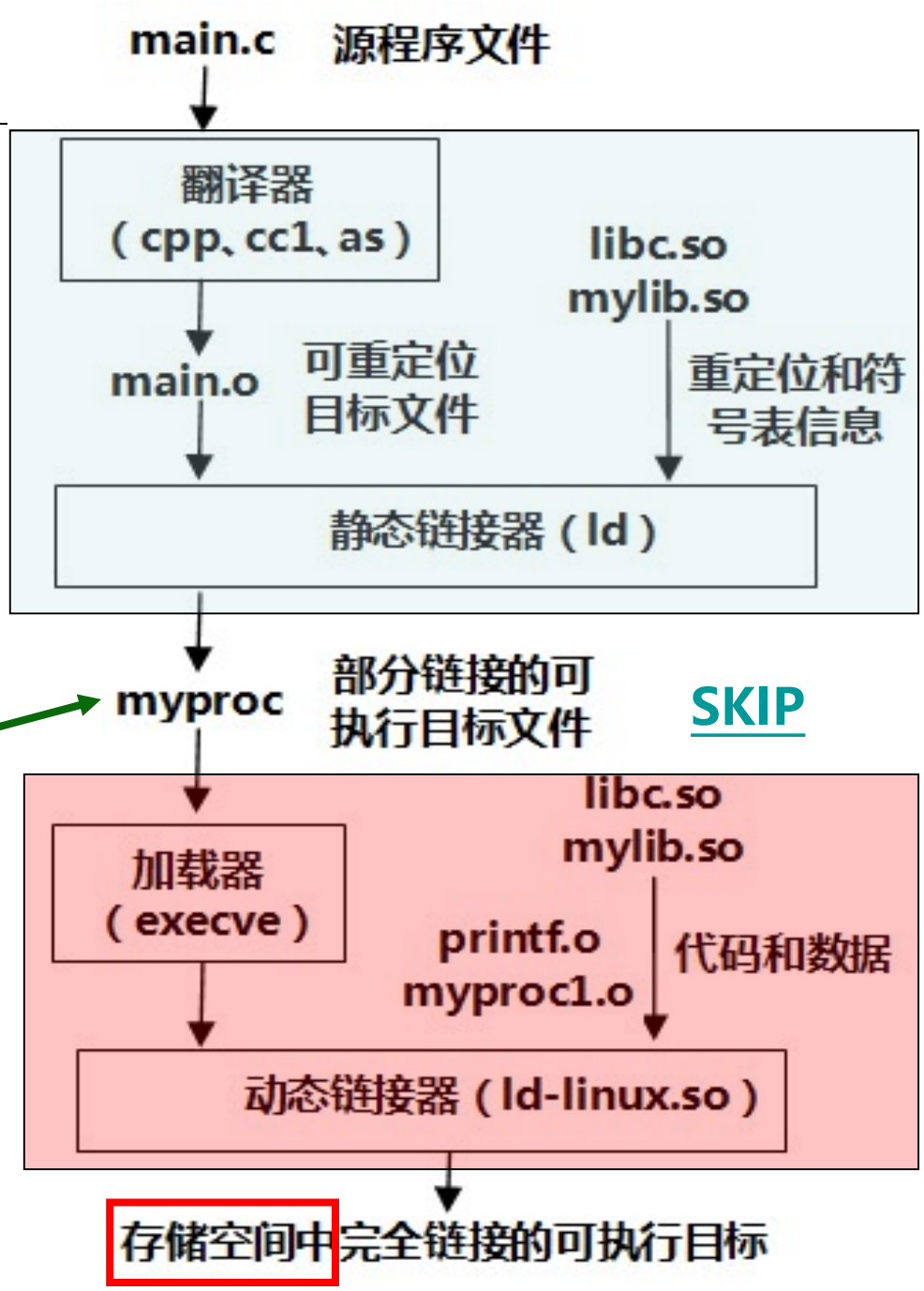
加载时动态链接

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。

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

BACK

```
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
```

运行时动态链

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

要实现动态链接，必须生成PIC代码

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+
0xfffffdb(-0x25)=
0x8048344

JMP指令也可用相对寻址方式解决

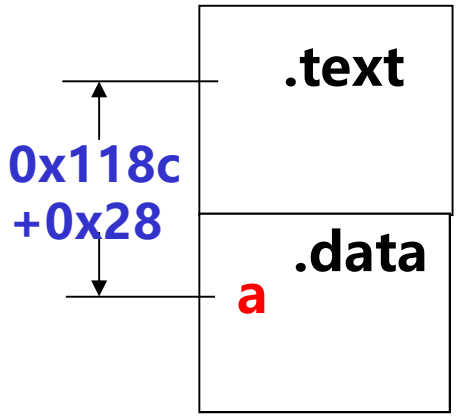
(2) 模块内部数据引用

- .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：

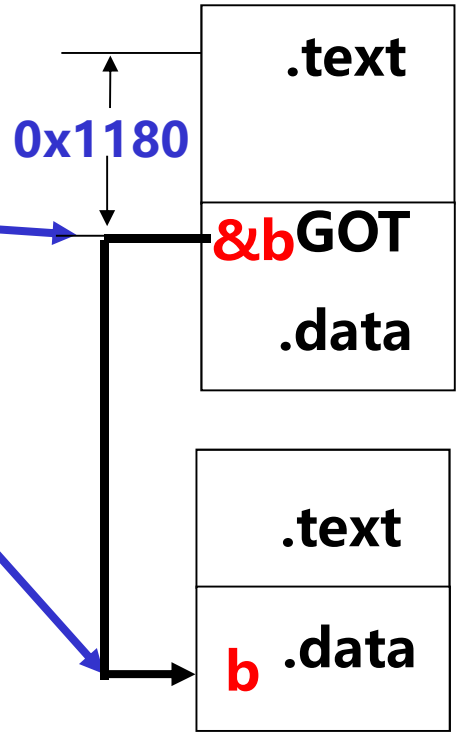
0x9000000+0x34c+0x118c(指令与.data间距离)+0x28(a在.data节中偏移)

(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;
}
......
```



共享库模块

- 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）

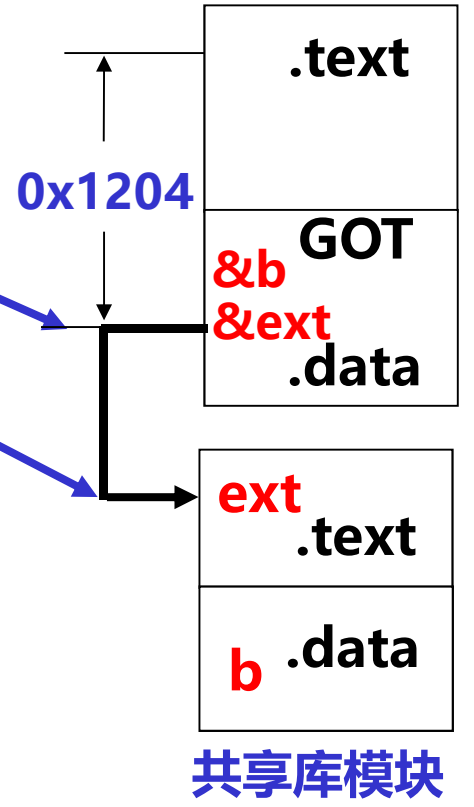
PIC有两个缺陷：多用4条指令；多了GOT（Global Offset Table），故需多用一个寄存器（如EBX），易造成寄存器溢出

(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

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

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

```
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]]
```



- 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）

(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>
```