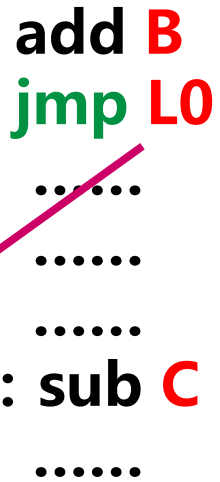
符号的重定位

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

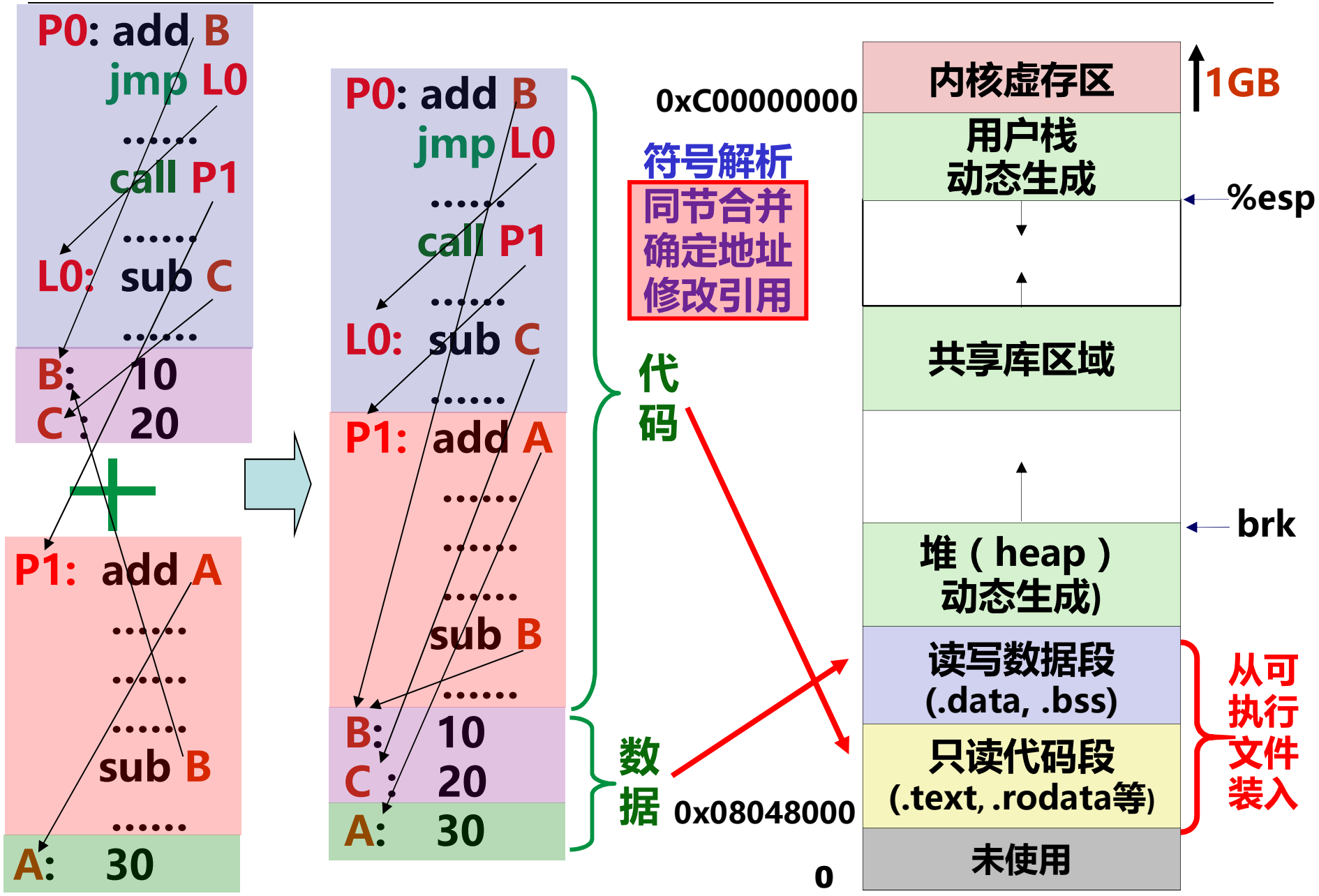
回顾：链接操作的步骤



- 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）

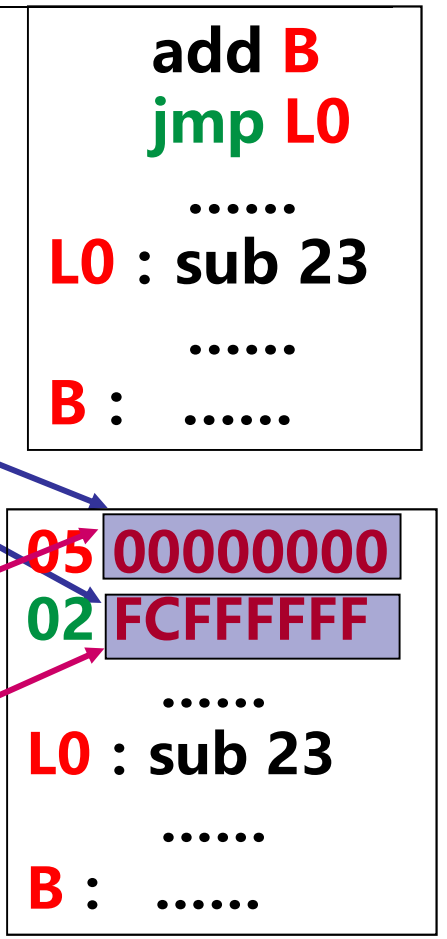
    需要用到在.rel_data和.rel_text节中保存的重定位信息

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！

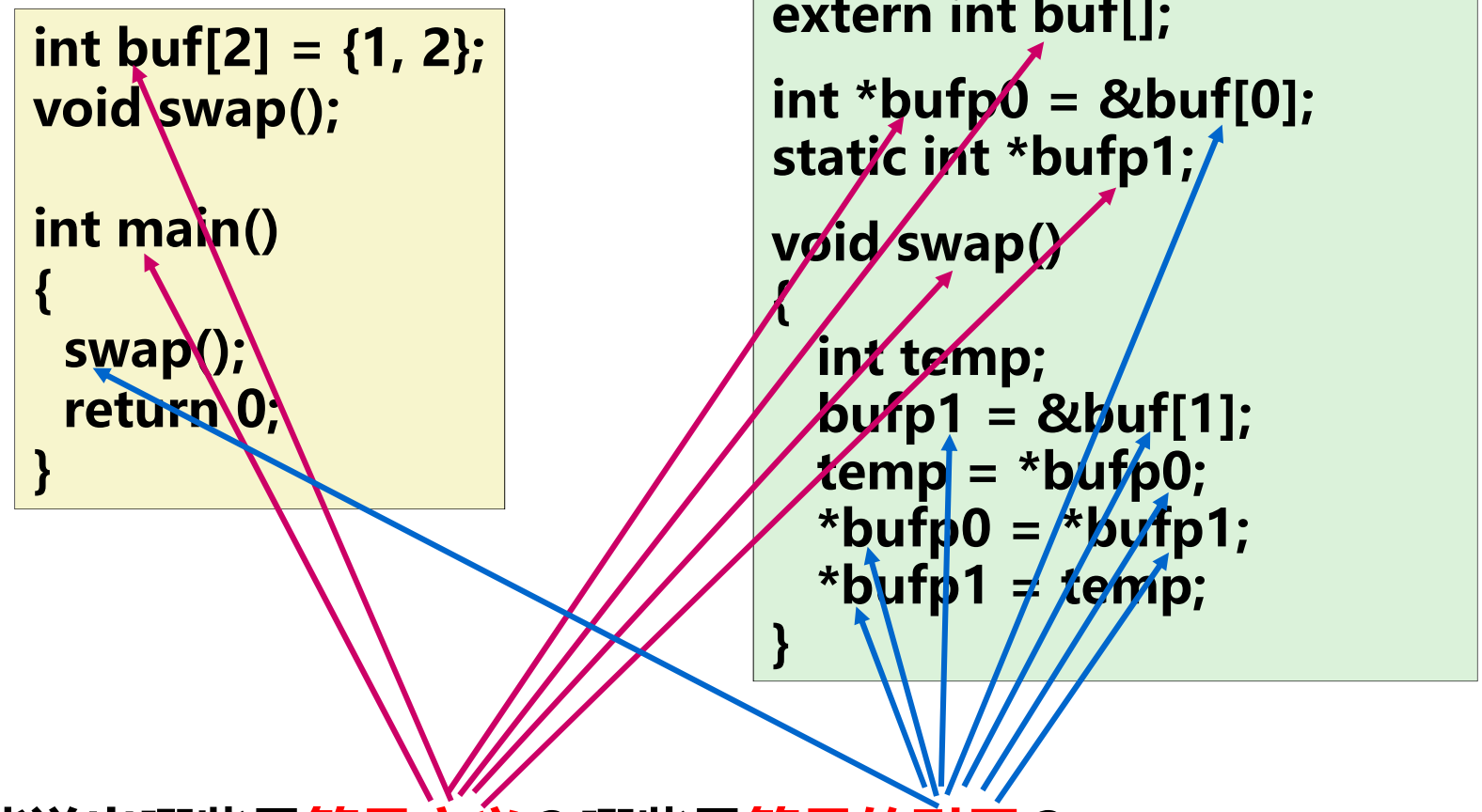
重定位操作举例

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？

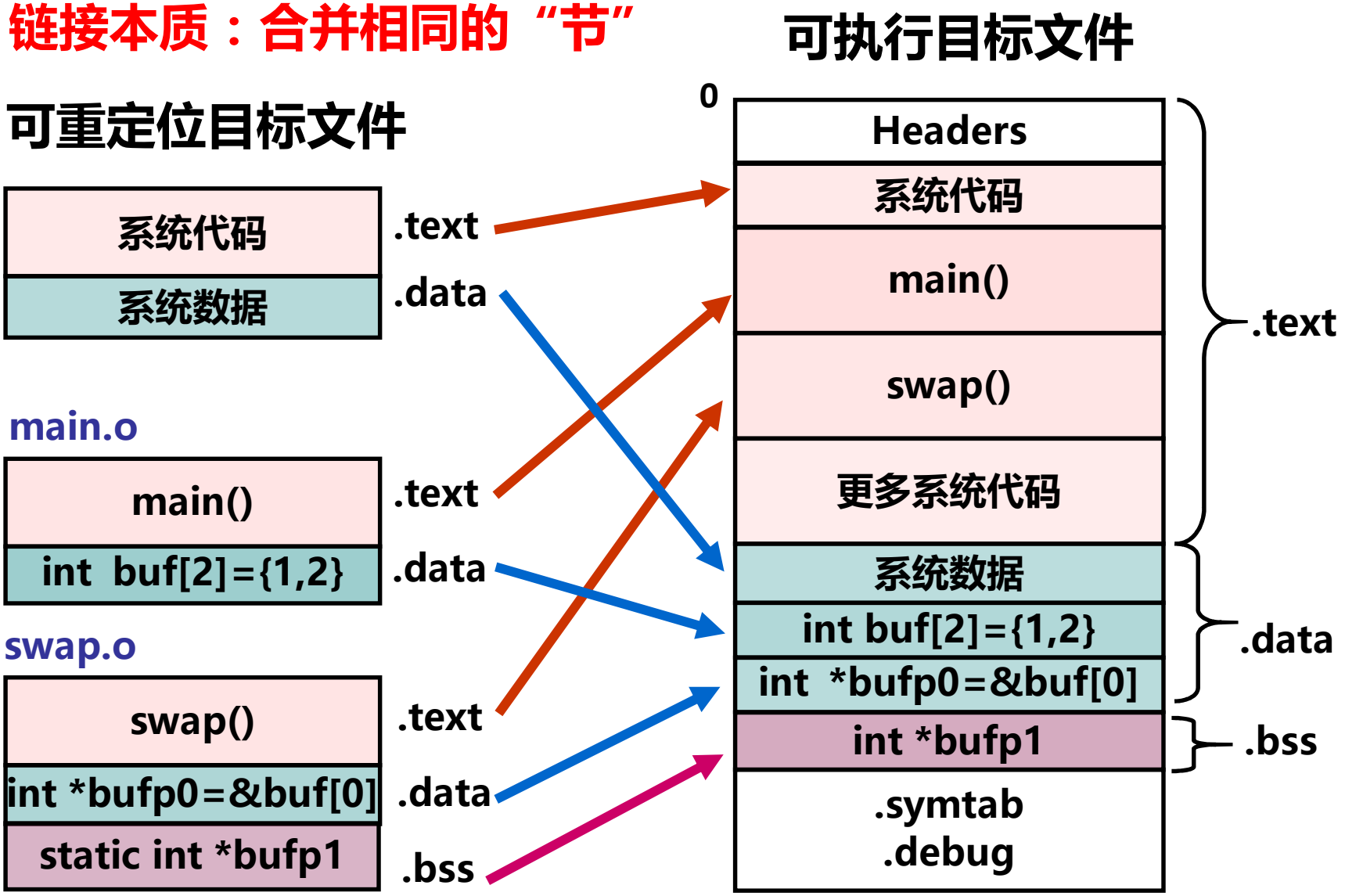
E中有main.o和swap.o两个模块！D中有所有定义的符号！

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

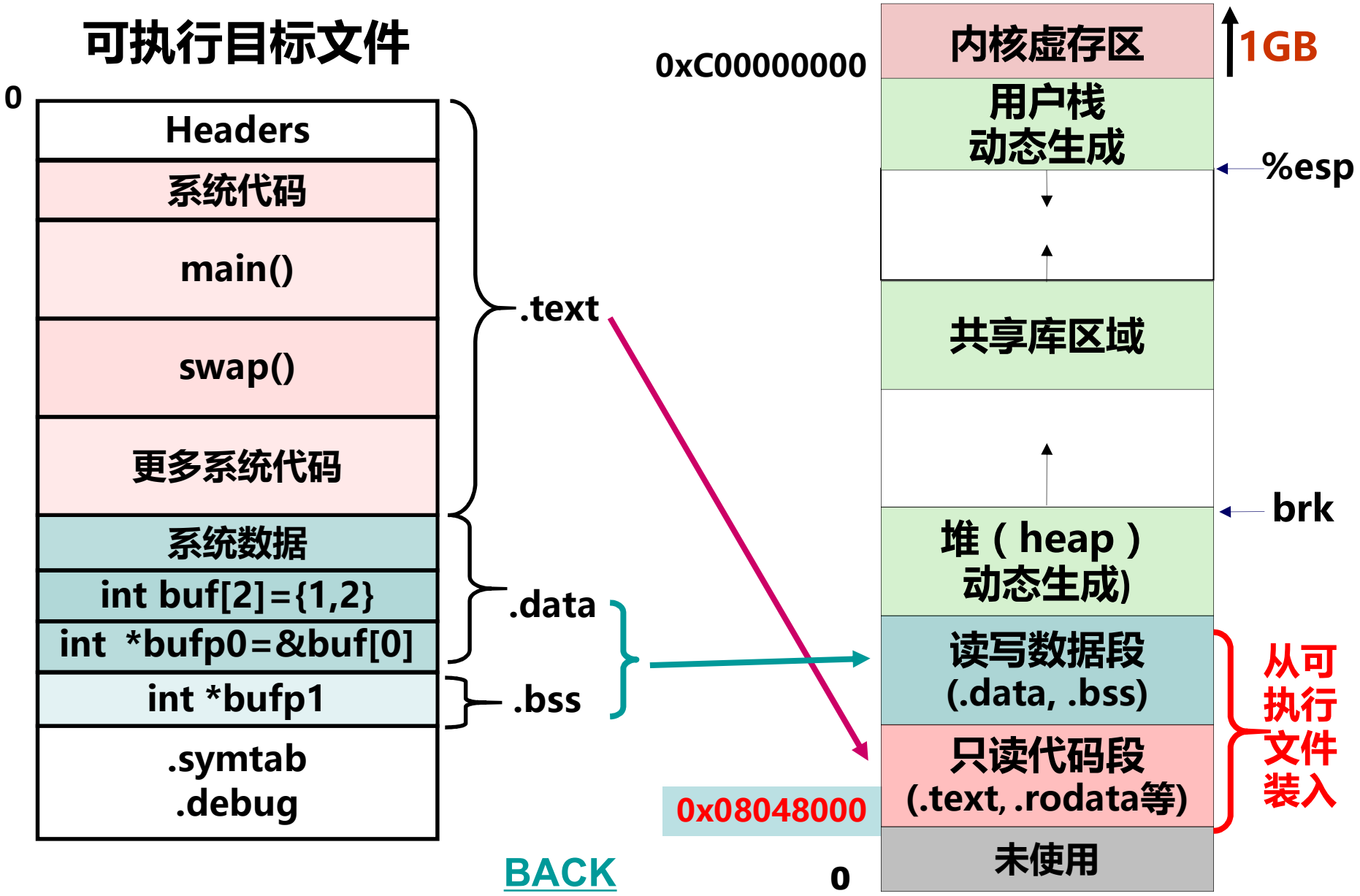
SKIP

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