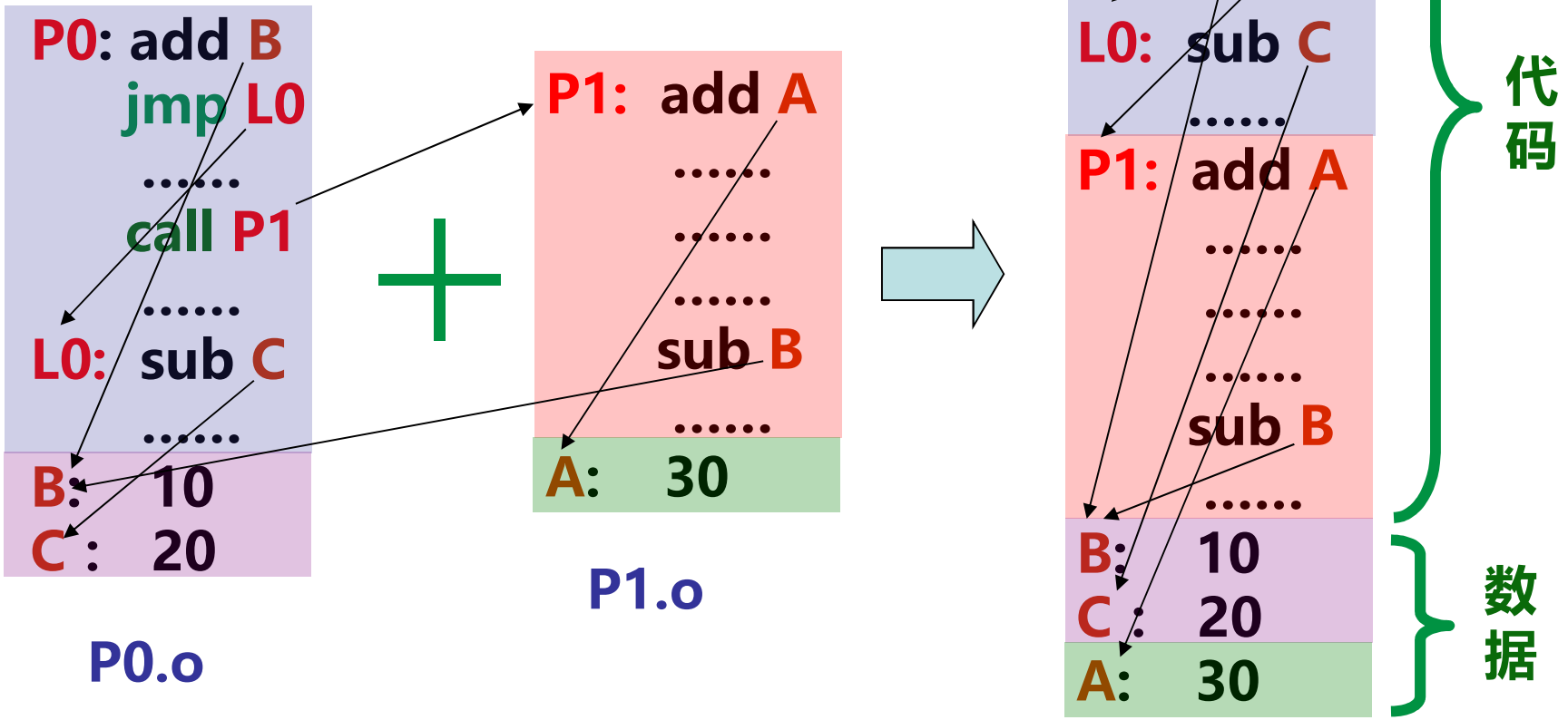
符号及符号表

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

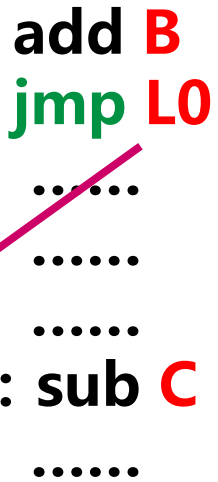
回顾：链接操作的步骤

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

重定位



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

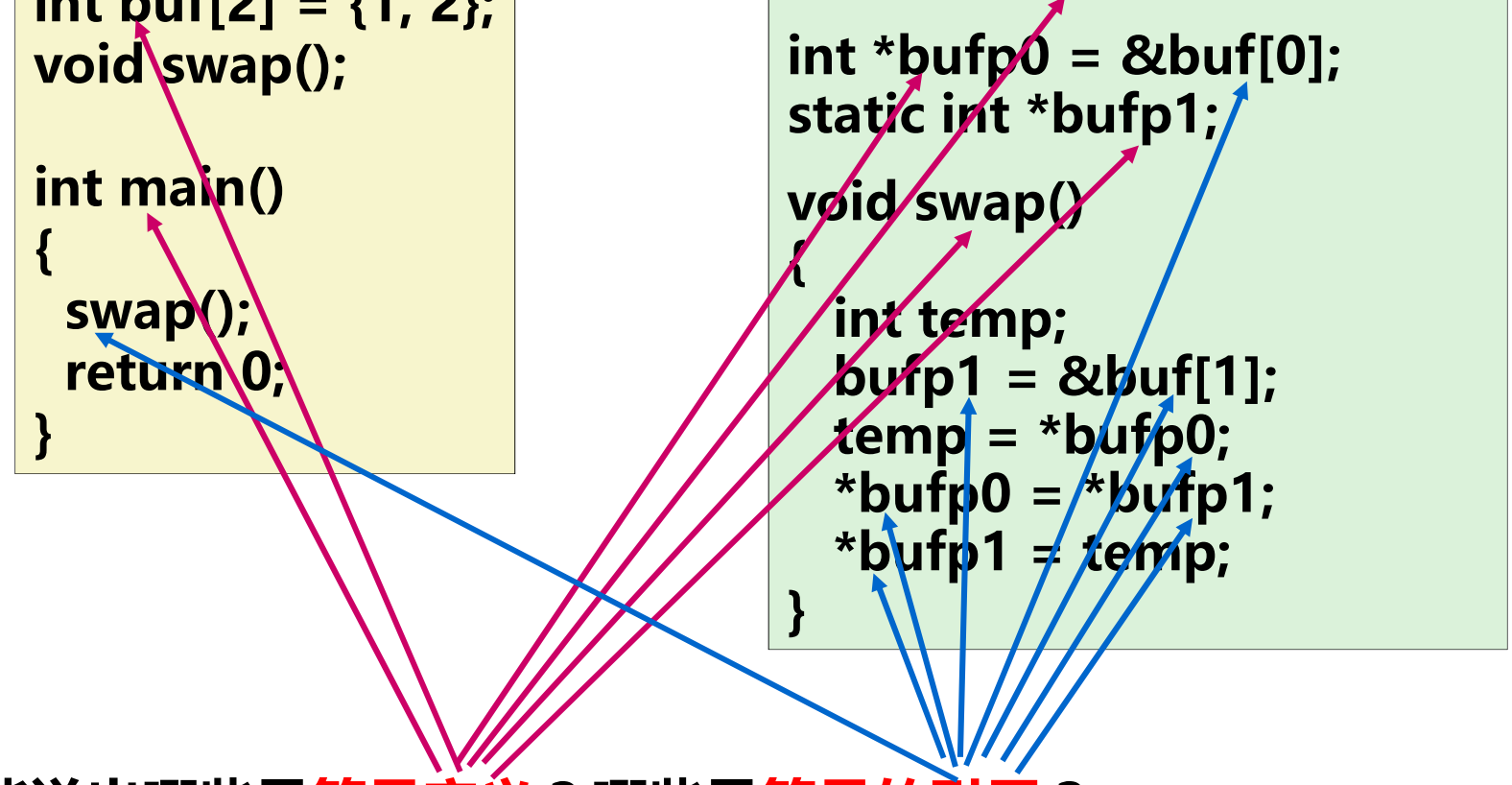
符号的定义和引用

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;
}
```



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

链接符号的类型

每个可重定位目标模块m都有一个符号表，它包含了在m中定义和引用的符号。有三种链接器符号：

- 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 C函数和非static的C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C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

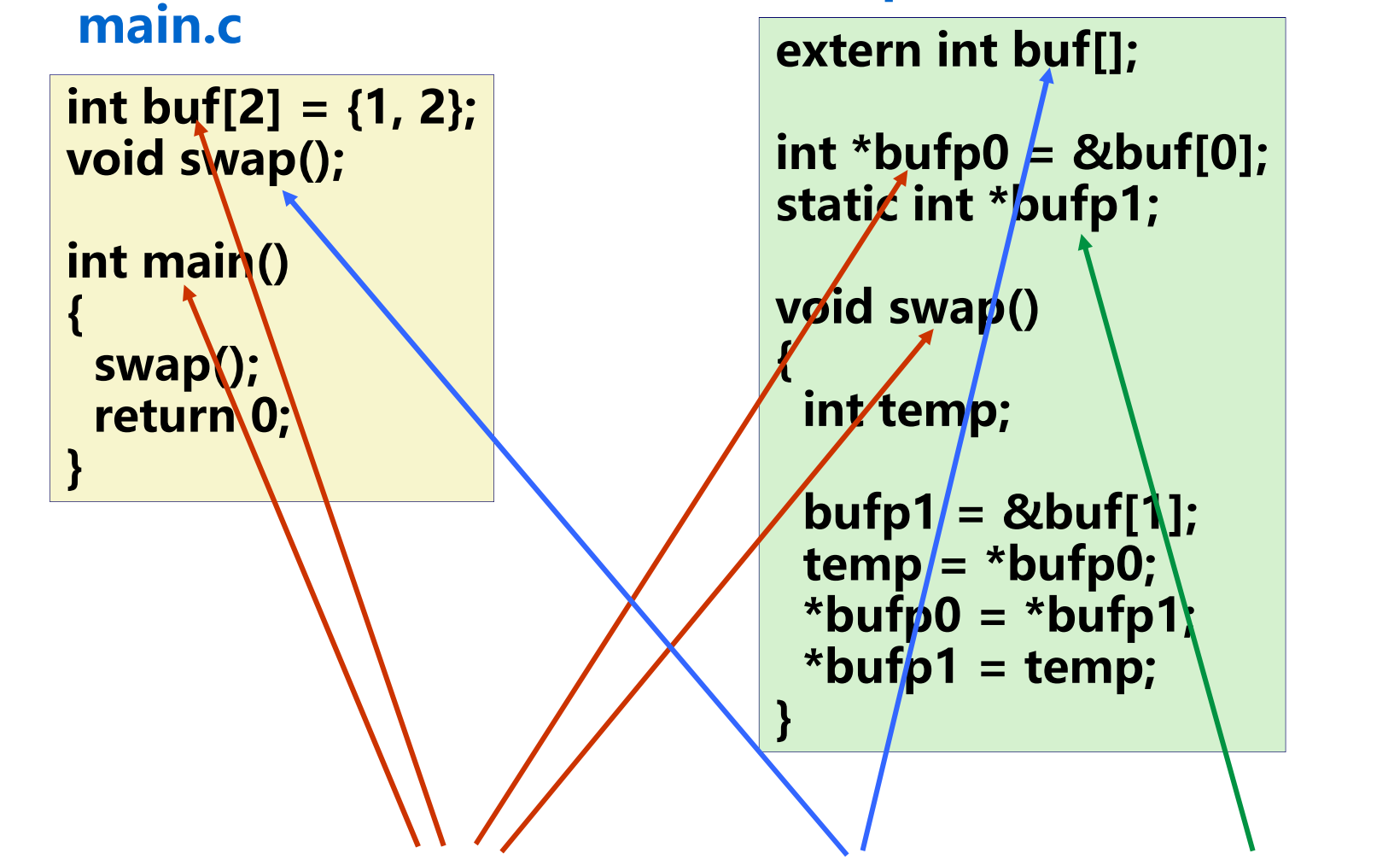
链接符号类型举例

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）中每个表项（16B）的结构如下：

函数名在text节中

变量名在data节或bss节中

```
typedef  struct {
    Elf32_Word    st_name;  /*符号对应字符串在strtab节中的偏移量*/
    Elf32_Addr    st_value;   /*在对应节中的偏移量，或虚拟地址*/
    Elf32_Word    st_size;     /*符号对应目标字节数*/
    unsigned char  st_info;   /*指出符号的类型(Type)和绑定属性(Bind) */
    unsigned char  st_other;
    Elf32_Half      st_shndx;  /*符号对应目标所在的节，或其他情况*/
} Elf32_Sym;
```

函数大小或变量长度

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

符号类型（Type）：数据、函数、源文件、节、未知

绑定属性（Bind）：全局符号、局部符号、弱符号

符号表信息举例

- 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	33	Func	Global	0	1	main
10:	0	0	Notype	Global	0	UND	swap

buf是main.o中第3节（.data）偏移为0的符号，是全局变量，占8B；
main是第1节（.text）偏移为0的符号，是全局函数，占33B；
swap是未定义的符号，不知道类型和大小，全局的（在其他模块定义）

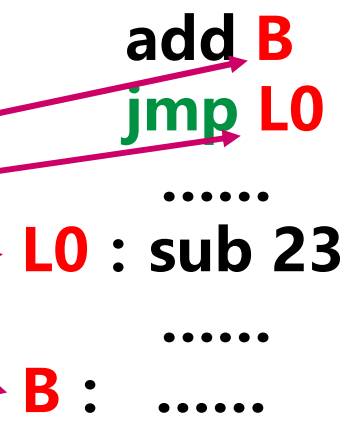
- swap.o中的符号表中最后4个条目（共11个）

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

bufp1是未分配地址且未初始化的本地变量(ndx=COM), 按4B对齐且占4B

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。



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？

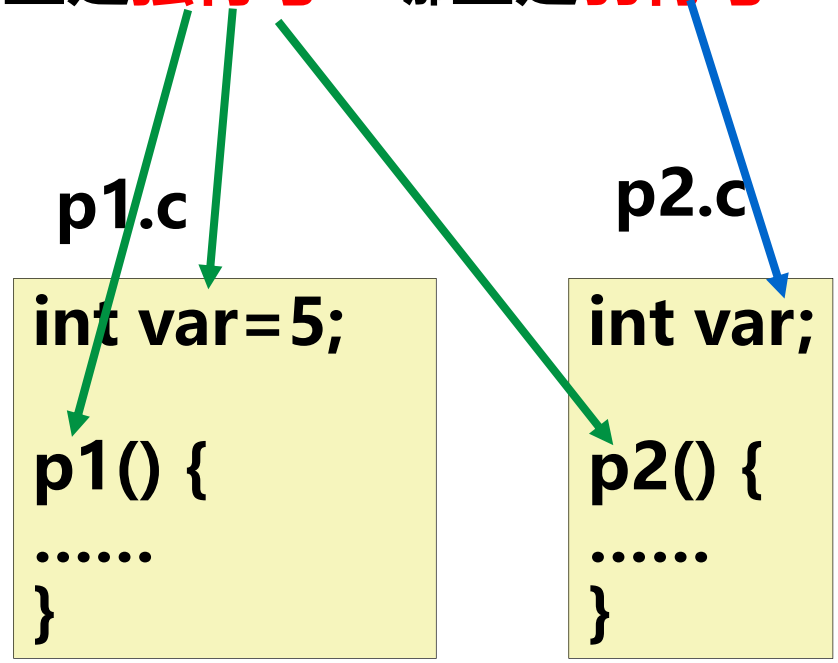
指被分配了存储空间。为函数名即指其代码所在区；为变量名即指其所占的静态数据区。

所有定义符号的值就是其目标所在的首地址

全局符号的强、弱

- 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？



全局符号的强、弱

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

本地局部符号

```
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

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。**

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）

多重定义符号的解析举例

以下程序会发生链接出错吗？

```
int  x=10;
int  p1(void);
int main()
{
     x=p1();
     return x;
}
```

main.c

```
int  x=20;
int p1()
{
     return x;
}
```

p1.c

main只有一次强定义

p1有一次强定义，一次弱定义

x有两次强定义，所以，链接器将输出一条出错信息

多重定义符号的解析举例

以下程序会发生链接出错吗？

```
# include <stdio.h>
int  y=100;
int  z;
void  p1(void);
int main()
{
    z=1000;
    p1( );
    printf( “y=%d, z=%d\n” , y, z);
    return 0;
}
```

main.c

y一次强定义，一次弱定义
z两次弱定义
p1一次强定义，一次弱定义
main一次强定义

```
int  y;
int  z;
void p1( )
{
     y=200;
     z=2000;
}
```

p1.c

问题：打印结果是什么？

y=200，z=2000

该例说明：在两个不同模块定义相同变量名，很可能发生意想不到的结果 ！

多重定义符号的解析举例

以下程序会发生链接出错吗？

```
1  #include <stdio.h>
2  int d=100;
3  int x=200;
4  void p1(void);
5  int main()
6  {
7     p1();
8     printf( "d=%d,x=%d\n" ,d,x);
9     return 0;
10 }
```

main.c

问题：打印结果是什么？

d=0,x=1 072 693 248

该例说明：两个重复定义的变量具有不同类型时，更容易出现难以理解的结果！

p1.c

```
1  double d;
2
3  void p1()
4  {
5     d=1.0;
6  }
```

FLD1
FSTPl &d

p1执行后d和x处内容是什么？

	0	1	2	3
&x	00	00	F0	3F
&d	00	00	00	00

1.0：0 01111111111 0…0B

=3FF0 0000 0000 0000H

多重定义符号的解析举例

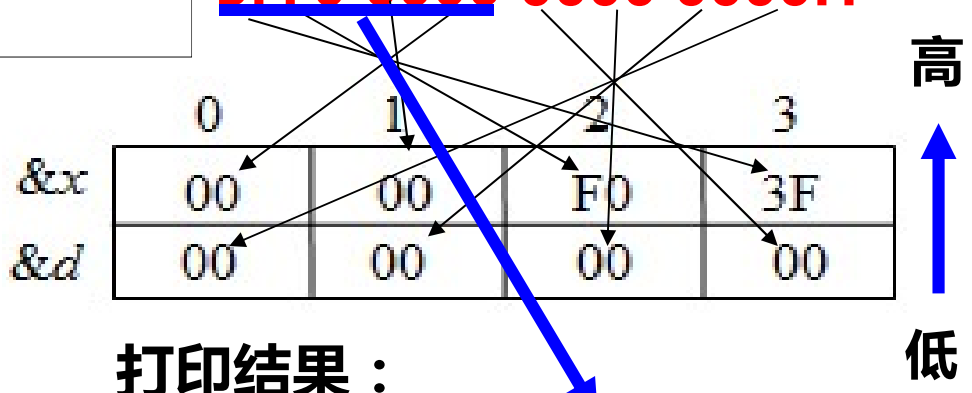
main.c

```
.......
1  int d=100;
2  int x=200;
3  int main()
4  {
5    p1( );
6    printf ( “d=%d, x=%d\n” , d, x );
7    return 0;
8  }
```

p1.c

```
1  double d;
2
3  void p1( )
4  {
5    d=1.0;
6  }
```

double型数1.0对应的机器数
3FF0 0000 0000 0000H



IA-32是小端方式

$2^{30}-1-(2^{20}-1)=2^{30}-2^{20}$

$=1024*1024*1023$

$=1\ 072\ 693\ 248$

打印结果：
d=0，x=1 072 693 248
Why？

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**

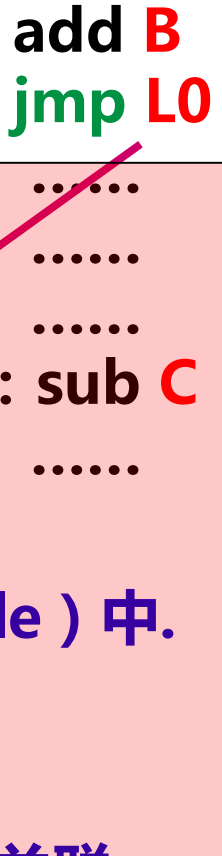
多重定义全局变量会造成一些意想不到的错误，而且是默默发生的，编译系统不会警告，并会在程序执行很久后才能表现出来，且远离错误引发处。特别是在一个具有几百个模块的大型软件中，这类错误很难修正。

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。

静态链接和符号解析

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

如何划分模块？

静态链接对象：

多个可重定位目标模块 + 静态库（标准库、自定义库）

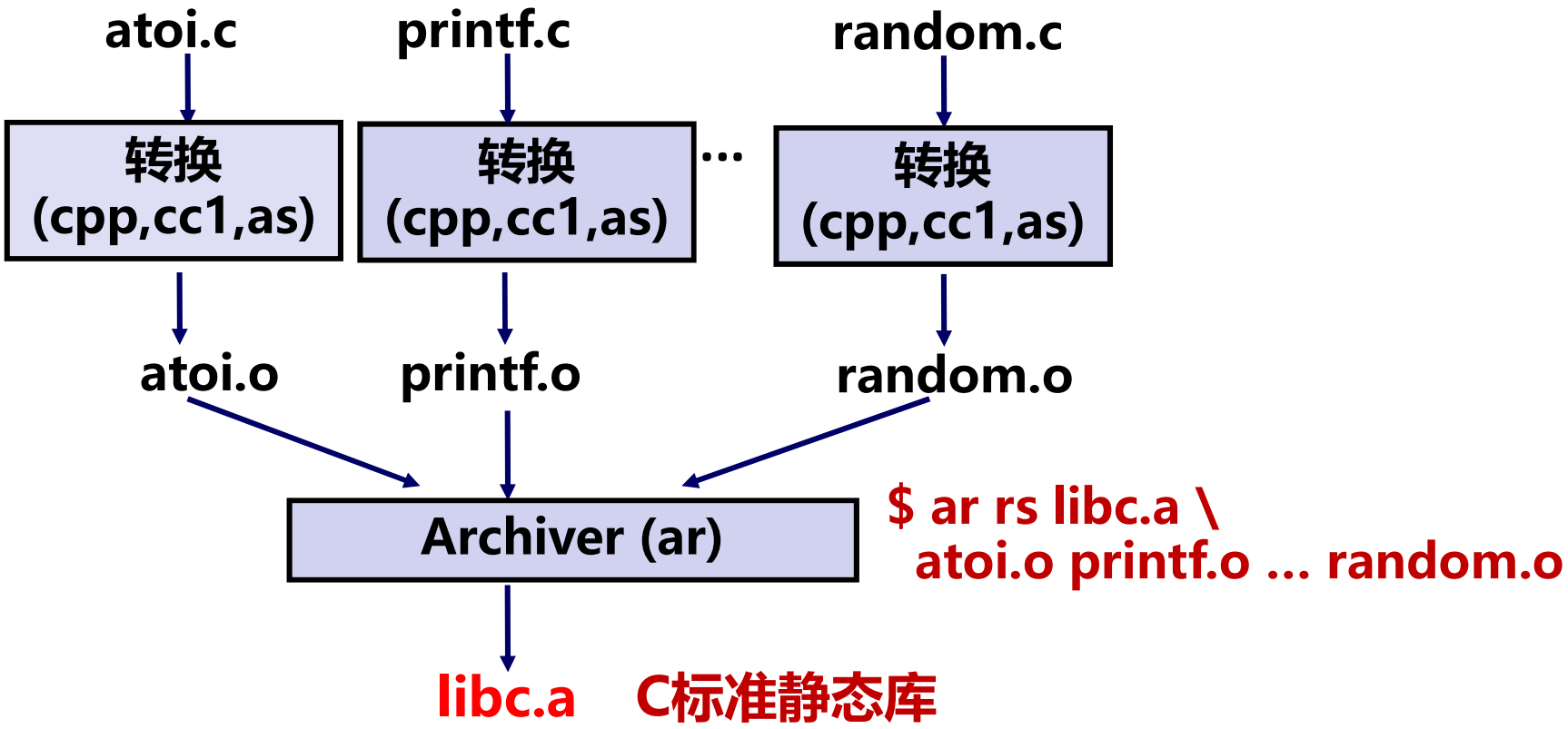
（.o文件）　　（.a文件，其中包含多个.o模块）

- **库函数模块：许多函数无需自己写，可使用共享的库函数**
  - **如数学库, 输入/输出库, 存储管理库，字符串处理等**
- **对于自定义模块，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）**
  - **使用静态库，可增强链接器功能，使其能通过查找一个或多个库文件中定义的符号来解析符号**
  - **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命令行中无需明显指定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
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
...
```

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

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！

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

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

libc.a无需明显指出！

调用关系：main→myfunc1→printf

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

main.c

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

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。因为模块myproc2.o没有被加入E中，因而它被丢弃。

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。

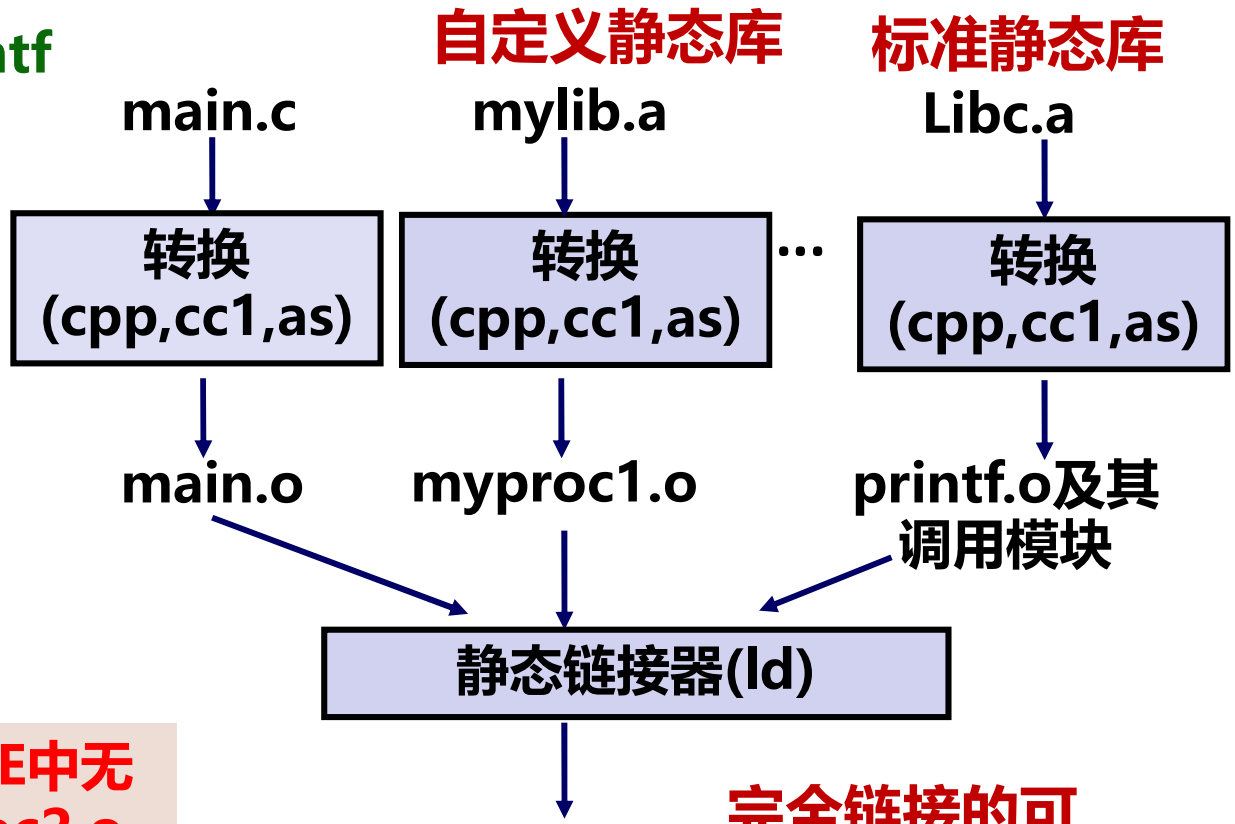
链接器中符号解析的全过程

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

main→myfunc1→printf

main.c

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



注意：E中无myproc2.o

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

链接器中符号解析的全过程

main.c

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

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

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及其引用符号

main→myfunc1→printf

被链接模块应按调用顺序指定！

若命令为：$ gcc –static –o myproc ./mylib.a main.o， 结果怎样？

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，故其中两个.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。Why？

因此，出现链接错误！

它只能用mylib.a中符号来解析，而mylib中两个.o模块都已被丢弃！

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

libmine.a 是静态库

假设调用关系：libtest.o→libfun.o(在libmine.a中）

-lxxx=libxxx.a (main) →(libfun)

```
$ gcc -L. libtest.o -lmine
$ gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

扫描libtest.o，将libfun送U，扫描到libmine.a时，用其定义的libfun来解析

说明在libtest.o中的main调用了libfun这个在库libmine中的函数，
所以，在命令行中，应该将libtest.o放在前面，像第一行中那样！

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

- 假设调用关系如下：

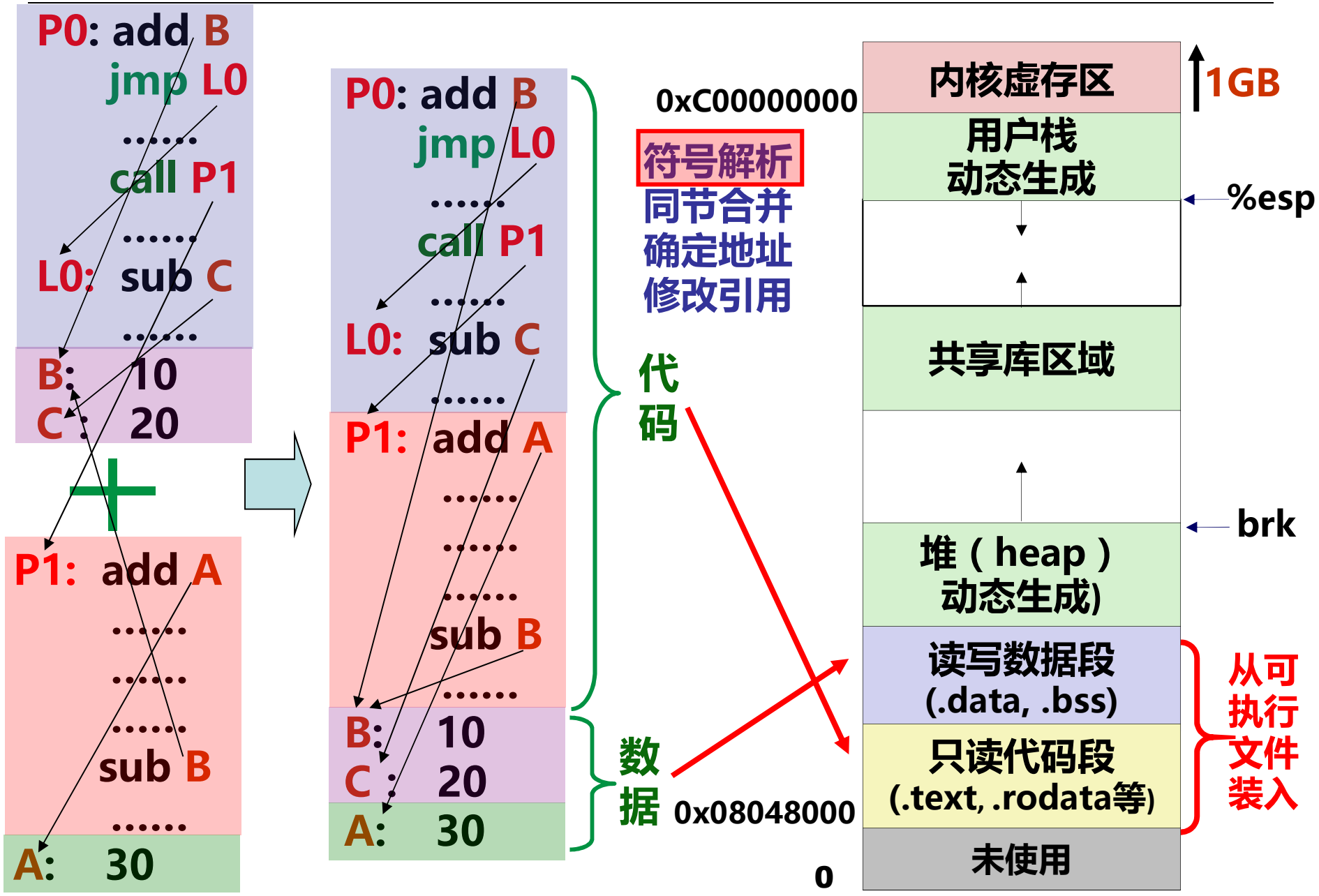
  func.o → libx.a 和 liby.a 中的函数

  libx.a → liby.a 同时 liby.a → libx.a

  则以下命令行可行：

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

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
%esp
共享库区域
堆（heap）
动态生成)
brk
读写数据段
(.data, .bss)
只读代码段
(.text, .rodata等)
未使用
1GB
从可执行文件装入