二进制基础

主讲人：immortal

CONTENTS

程序员的自我修养
——链接、装载与库

俞甲子 石凡 潘爱民 著

電子工業出版社

1 从hello world 开始

test.cpp

```cpp
#include <stdio.h>

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

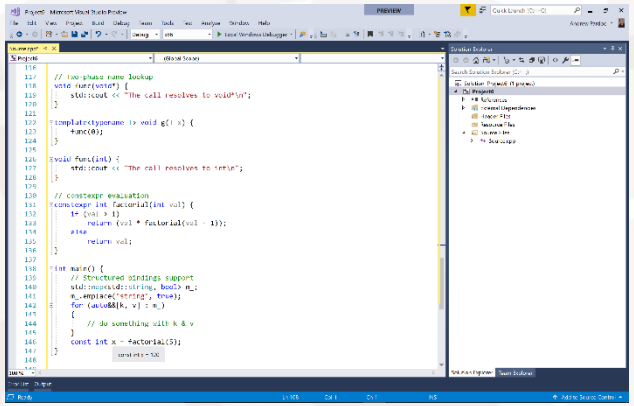
问题 输出 终端 端口 调试控制台

```
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test# gcc test.cpp
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test# ./a.out
Hello world
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test#
```

1 编译

1 编译

GNU Compiler Collection



Windows compiler (cl.exe)

Clang

1 编译

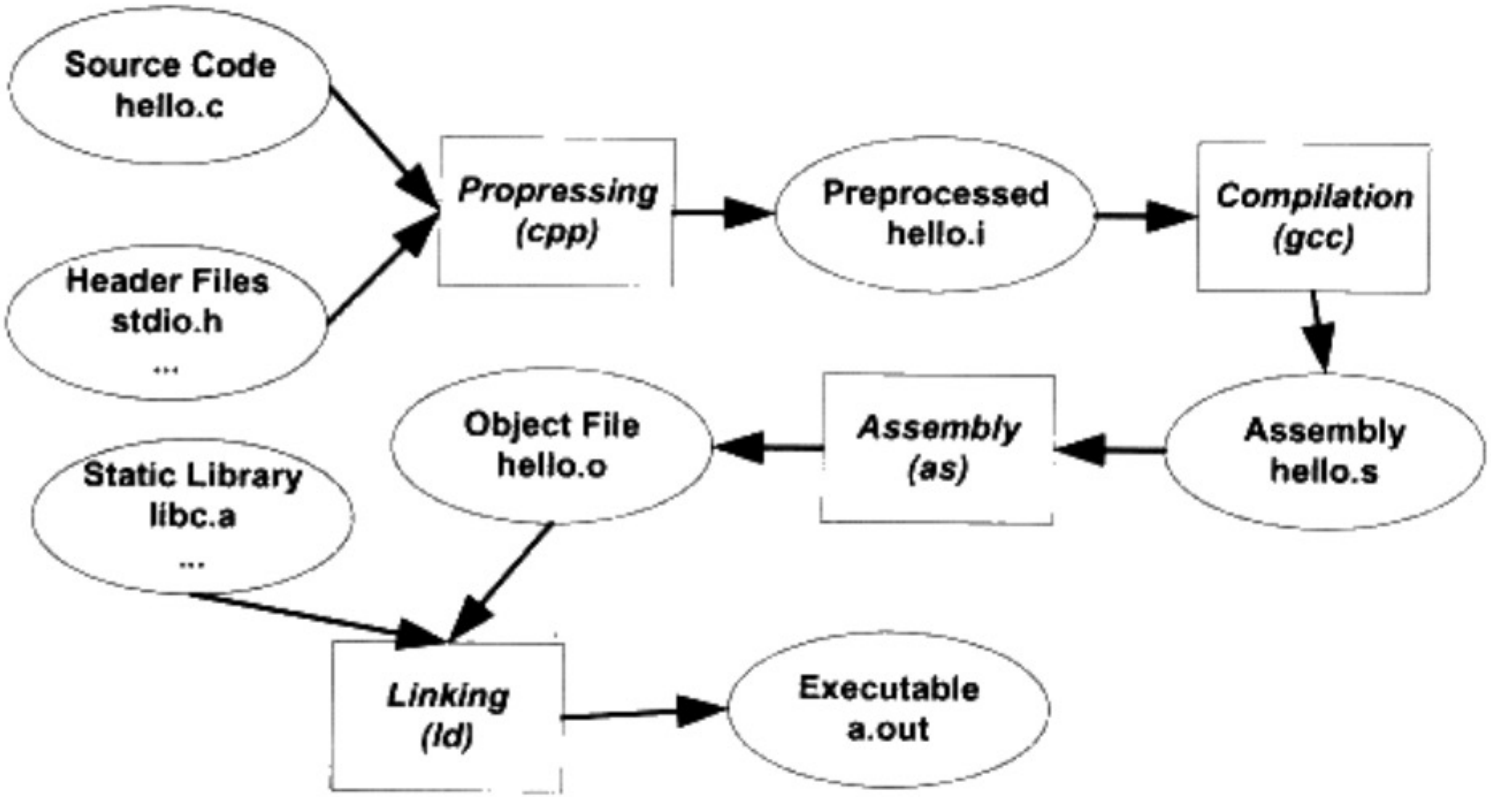


图 2-1　GCC 编译过程分解

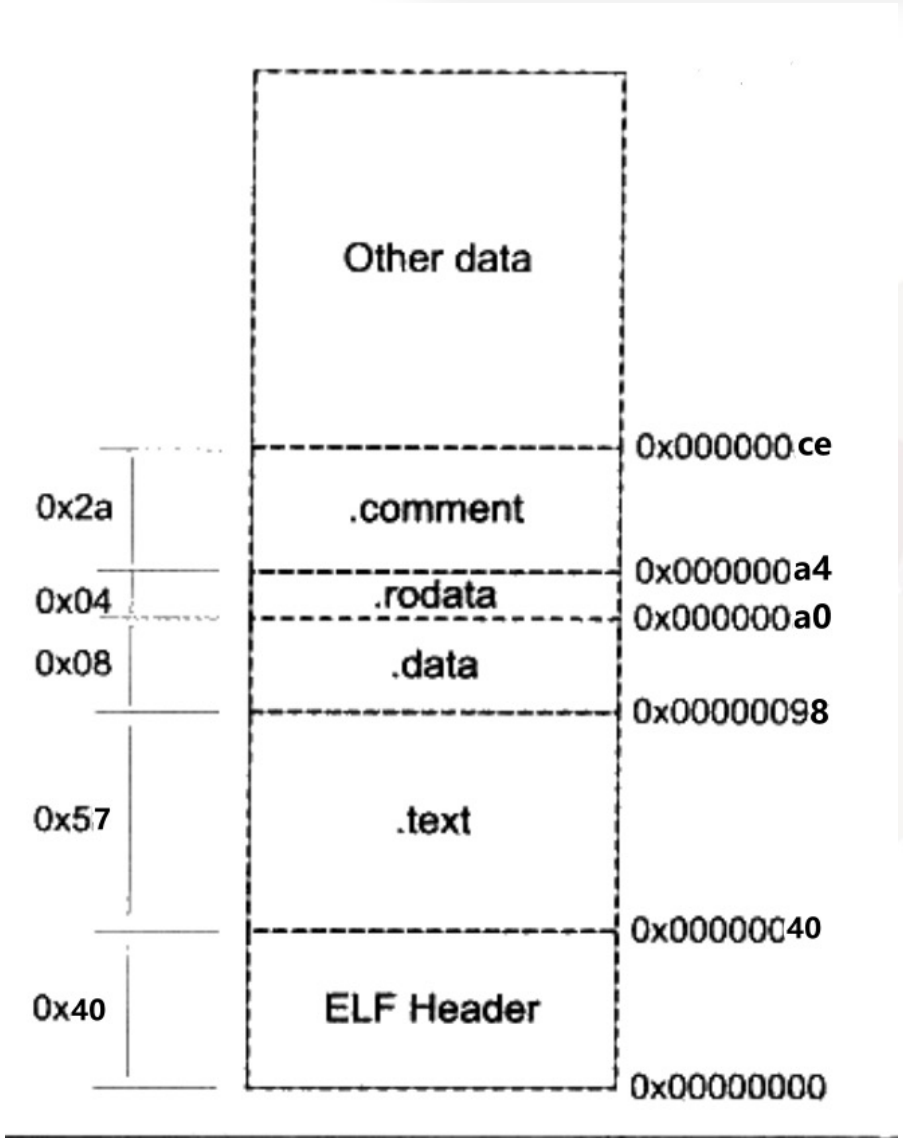
2 ELF文件

2 ELF文件

```
int printf(const char* format, ...);
int global_innit_var = 1;
int global_uninit_var;
void func(int i){
    printf("%d\n",i);
}
int main(){
    static int static_var = 2;
    static int static_var2;
    int a=1;
    int b;

func(static_var+static_var2+a+b);
    return a;
}
```

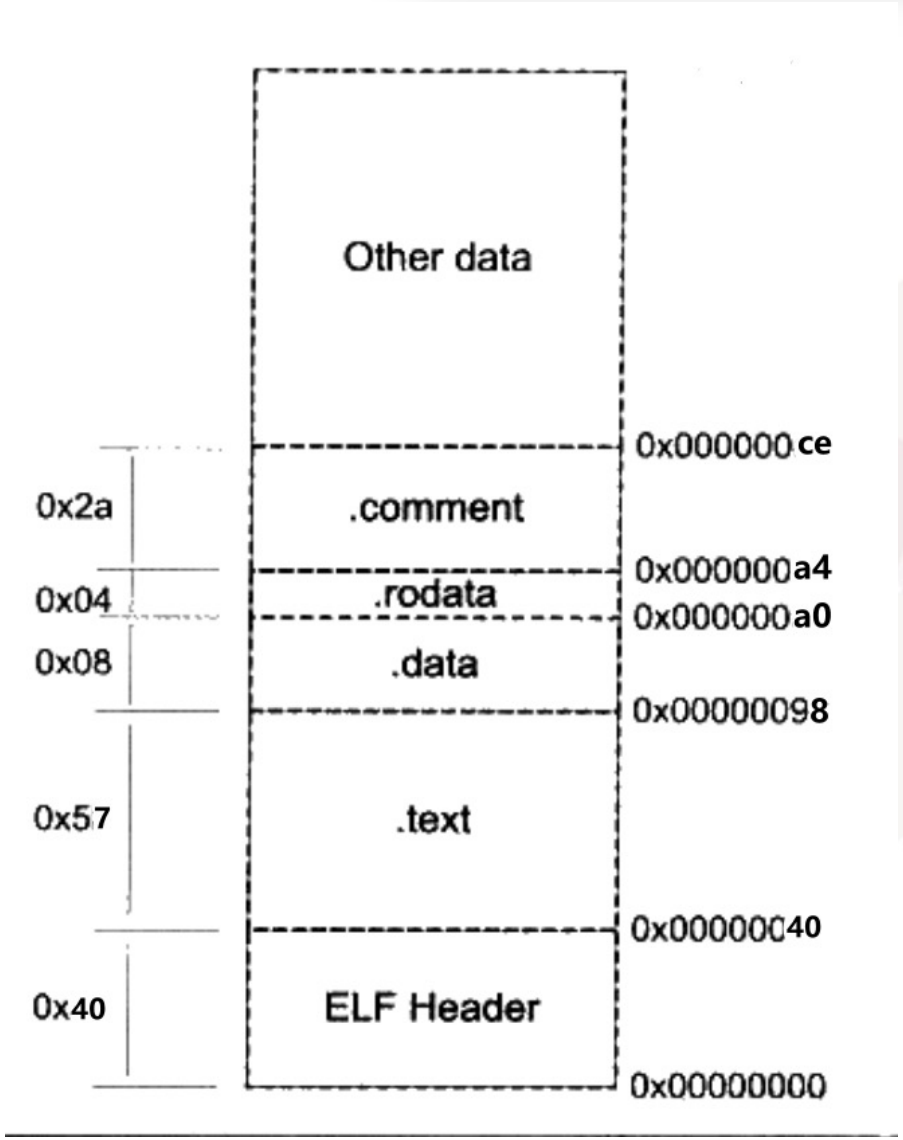
```
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test# gcc -c test.c
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test# objdump -h test.o

test.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000057  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000008  0000000000000000  0000000000000000  00000098  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  0000000000000000  0000000000000000  000000a0  2**2
                  ALLOC
  3 .rodata       00000004  0000000000000000  0000000000000000  000000a0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000002a  0000000000000000  0000000000000000  000000a4  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000ce  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000d0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test#
```

2 ELF文件 .text段




```
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test# objdump -d test.o

test.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <func>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   8b 45 fc                mov    -0x4(%rbp),%eax
   e:   89 c6                   mov    %eax,%esi
  10:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 17 <func+0x17>
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   e8 00 00 00 00          callq  21 <func+0x21>
  21:   90                      nop
  22:   c9                      leaveq
  23:   c3                      retq

0000000000000024 <main>:
  24:   55                      push   %rbp
  25:   48 89 e5                mov    %rsp,%rbp
  28:   48 83 ec 10             sub    $0x10,%rsp
  2c:   c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
  33:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 39 <main+0x15>
  39:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 3f <main+0x1b>
  3f:   01 c2                   add    %eax,%edx
  41:   8b 45 f8                mov    -0x8(%rbp),%eax
  44:   01 c2                   add    %eax,%edx
  46:   8b 45 fc                mov    -0x4(%rbp),%eax
  49:   01 d0                   add    %edx,%eax
  4b:   89 c7                   mov    %eax,%edi
  4d:   e8 00 00 00 00          callq  52 <main+0x2e>
  52:   8b 45 f8                mov    -0x8(%rbp),%eax
  55:   c9                      leaveq
  56:   c3                      retq
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test#
```

.data和.rodata 段

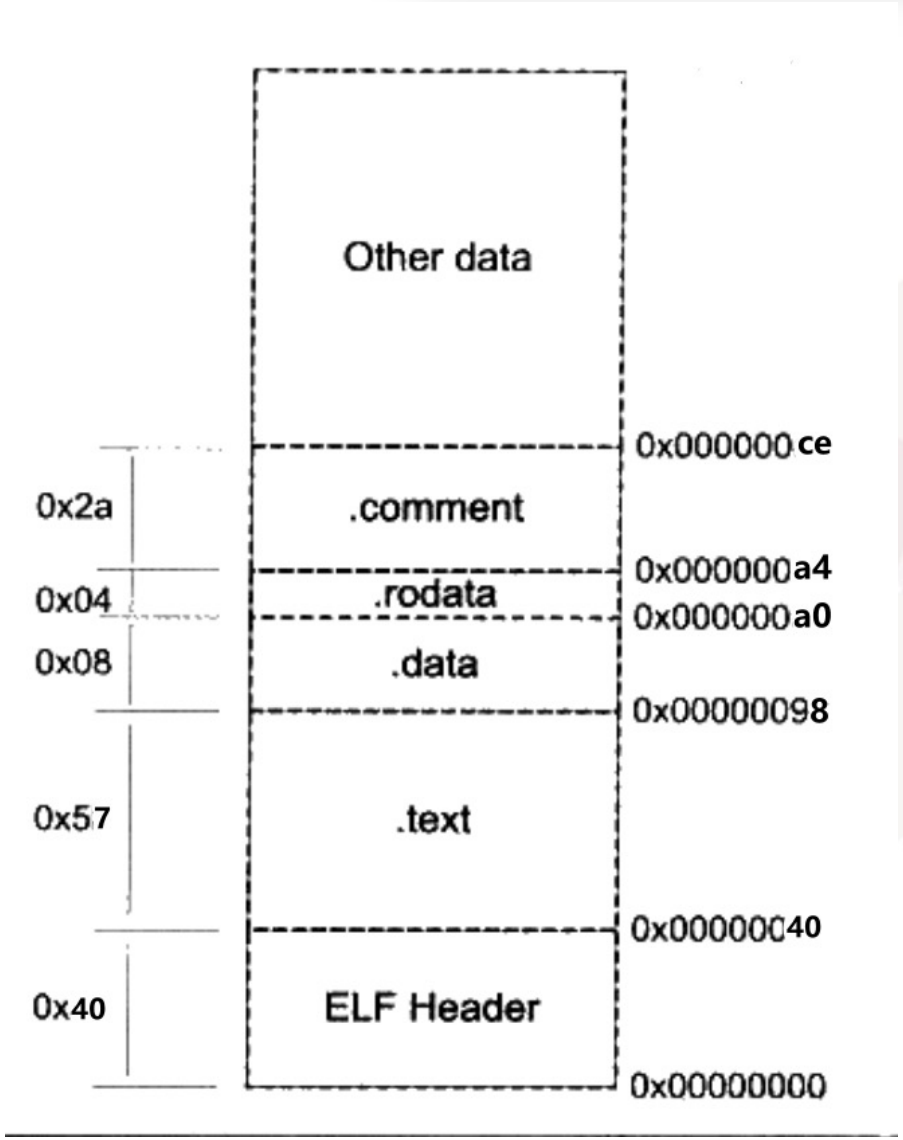


.data 段保存了**已经初始化的全局静态变量**和**局部静态变量。** 即前面程序中的global_init_var和static_var.

.rodata 段保存了只读的数据，一般是程序里的只读变量（const修饰）和字符串常量。 即前面程序中的字符串"%d\n"。

```
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test# hexdump -C test.o
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  48 04 00 00 00 00 00 00  |........H.......|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 0d 00 0c 00  |....@.....@.....|
00000040  55 48 89 e5 48 83 ec 10  89 7d fc 8b 45 fc 89 c6  |UH..H....}..E...|
00000050  48 8d 3d 00 00 00 00 b8  00 00 00 00 e8 00 00 00  |H.=.............|
00000060  00 90 c9 c3 55 48 89 e5  48 83 ec 10 c7 45 f8 01  |....UH..H....E..|
00000070  00 00 00 8b 15 00 00 00  00 8b 05 00 00 00 00 01  |................|
00000080  c2 8b 45 f8 01 c2 8b 45  fc 01 d0 89 c7 e8 00 00  |..E....E........|
00000090  00 00 8b 45 f8 c9 c3 00  01 00 00 00 02 00 00 00  |...E............|
000000a0  25 64 0a 00 00 47 43 43  3a 20 28 55 62 75 6e 74  |%d...GCC: (Ubunt|
000000b0  75 20 37 2e 35 2e 30 2d  33 75 62 75 6e 74 75 31  |u 7.5.0-3ubuntu1|
000000c0  7e 31 38 2e 30 34 29 20  37 2e 35 2e 30 00 00 00  |~18.04) 7.5.0...|
```

.bss段



.bss 段保存了**未初始化的全局静态变量**和**局部静态变量。** 即前面程序中的global_uninit_var和static_var2.

.bss只有4字节，我们有两个int变量应该有8字节，这是为什么？

objdump -x test.o

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000004 l     O .data  0000000000000004 static var.1801
0000000000000000 l     O .bss   0000000000000004 static_var2.1802
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     O .data  0000000000000004 global innit var
0000000000000004       O *COM*  0000000000000004 global uninit var
0000000000000000 g     F .text  0000000000000024 func
0000000000000000         *UND*  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*  0000000000000000 printf
0000000000000024 g     F .text  0000000000000033 main
```

func.c

```
int buf = 0;

void func() {
  buf = 2;
}
```

main.c

```
#include <stdio.h>
int buf;
void func();

int main() {
  buf = 1;
  func();
  printf("%d\n", buf);
  return 0;
}
```

C语言编译器将全局符号标记为strong和weak两类：

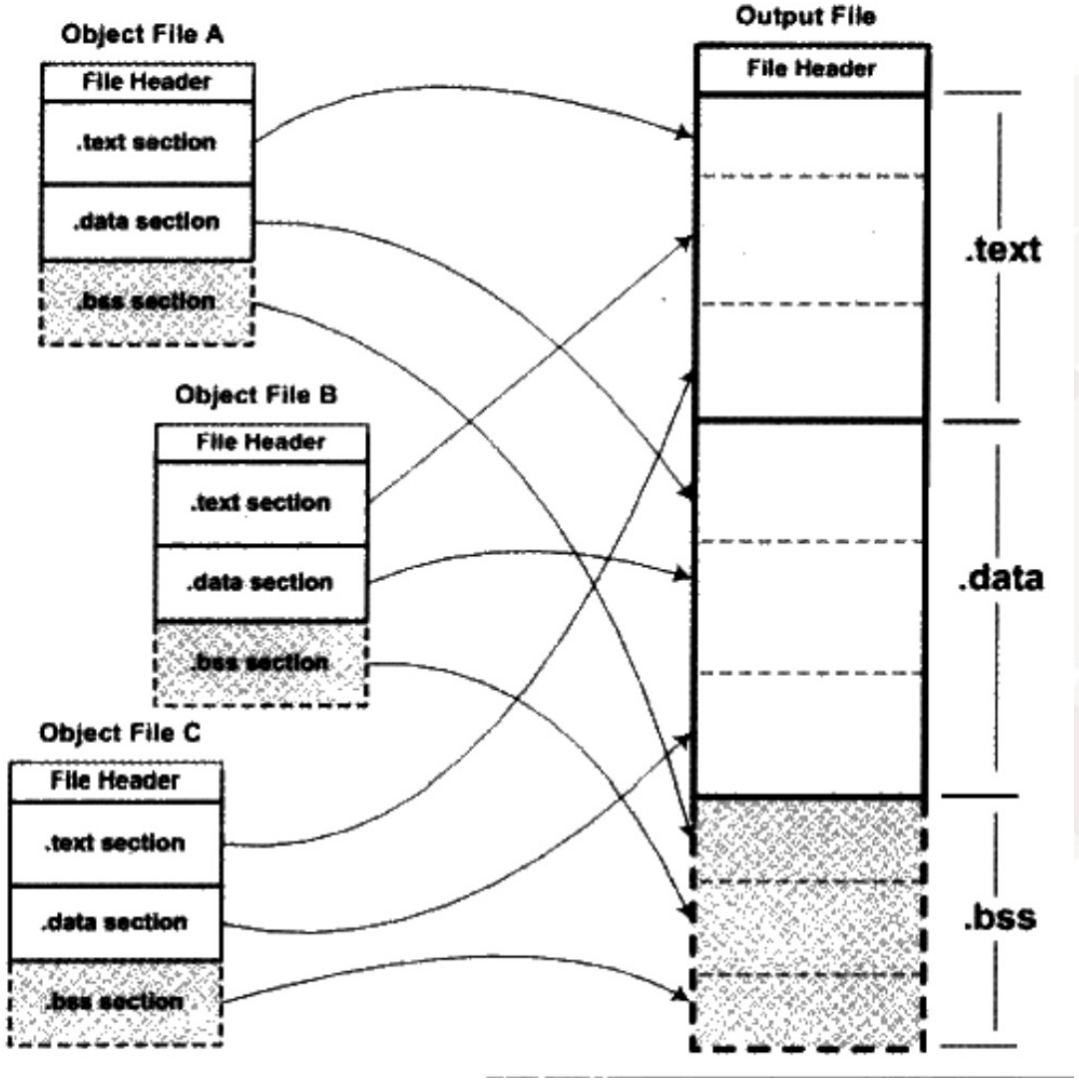
- 函数和初始化的全局符号被标记为strong
- 未初始化的全局符号被标记为weak

其他段

常用的段名	说明
.rodata1	Read only Data，这种段里存放的是只读数据，比如字符串常量、全局 const 变量。跟“.rodata”一样
.comment	存放的是编译器版本信息，比如字符串：”GCC: (GNU) 4.2.0”
.debug	调试信息
.dynamic	动态链接信息
.hash	符号哈希表
.line	调试时的行号表，即源代码行号与编译后指令的对应表
.note	额外的编译器信息。比如程序的公司名、发布版本号等
.strtab	String Table.字符串表，用于存储 ELF 文件中用到的各种字符串
.symtab	Symbol Table.符号表
.shstrtab	Section String Table.段名表
.plt .got	动态链接的跳转表和全局入口表
.init .fini	程序初始化与终结代码段。

3 静态链接

3 静态链接



Step 1：空间与地址分配

- 扫描所有的输入目标文件,获得它们的各个段的各种属性
- 收集所有目标文件中的符号定义和符号引用到一个全局符号表
- 合并所有段并建立映射关系

Step 2：符号解析和重定位

- 通过step1中收集的信息,读取输入文件中段的数据和重定位信息,进行符号解析和重定位

3 静态链接

空间与地址分配

a.c

```
extern int shared;
int main(){
    int a = 100;
    swap(&a,&shared);
}
```

b.c

```
int shared = 1;
void swap(int *a , int *b){
    *a ^= *b ^= *a ^= *b ;
}
```

链接之前,目标文件中的所有段的VMA(Virtual Memory Address)都是0,因为虚拟空间还没有被分配

链接之后, 可执行文件ab中的各个段都被分配到了相应的虚拟地址

```
(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
$ gcc a.c b.c
a.c: In function ‘main’:
a.c:4:5: warning: implicit declaration of function ‘swap’ [-Wimplicit-function-declara
    4 |     swap(&a,&shared);
      |     ^~~~

(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
$ ld a.o b.o -e main -o ab

(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
$ objdump -h a.o

a.o:     file format elf64-x86-64

                          Virtual Memory Address
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000002e  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  0000006e  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  0000006e  2**0
                  ALLOC
  3 .comment      0000001f  0000000000000000  0000000000000000  0000006e  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  0000008d  2**0
                  CONTENTS, READONLY
  5 .eh_frame     00000038  0000000000000000  0000000000000000  00000090  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA

(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
$ objdump -h b.o

b.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000004b  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  0000000000000000  0000000000000000  0000008c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000090  2**0
                  ALLOC
  3 .comment      0000001f  0000000000000000  0000000000000000  00000090  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000af  2**0
                  CONTENTS, READONLY
  5 .eh_frame     00000038  0000000000000000  0000000000000000  000000b0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA

(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
$ objdump -h ab

ab:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000079  0000000000401000  0000000000401000  00001000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .eh_frame     00000058  0000000000402000  0000000000402000  00002000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000004  0000000000404000  0000000000404000  00003000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .comment      0000001e  0000000000000000  0000000000000000  00003004  2**0
                  CONTENTS, READONLY
```

3 静态链接

重定位

```
┌──(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
└─$ objdump -d a.o

a.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 fc 64 00 00 00    movl   $0x64,-0x4(%rbp)   a = 100
   f:   48 8d 45 fc             lea    -0x4(%rbp),%rax
  13:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 1a <main+0x1a>
  1a:   48 89 c7                mov    %rax,%rdi
  1d:   b8 00 00 00 00          mov    $0x0,%eax
  22:   e8 00 00 00 00          call   27 <main+0x27>
  27:   b8 00 00 00 00          mov    $0x0,%eax
  2c:   c9                      leave
  2d:   c3                      ret
```

```
┌──(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
└─$ objdump -d ab

ab:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <main>:
  401000:       55                      push   %rbp
  401001:       48 89 e5                mov    %rsp,%rbp
  401004:       48 83 ec 10             sub    $0x10,%rsp
  401008:       c7 45 fc 64 00 00 00    movl   $0x64,-0x4(%rbp)
  40100f:       48 8d 45 fc             lea    -0x4(%rbp),%rax
  401013:       48 8d 35 e6 2f 00 00    lea    0x2fe6(%rip),%rsi        # 404000 <shared>
  40101a:       48 89 c7                mov    %rax,%rdi
  40101d:       b8 00 00 00 00          mov    $0x0,%eax
  401022:       e8 07 00 00 00          call   40102e <swap>
  401027:       b8 00 00 00 00          mov    $0x0,%eax
  40102c:       c9                      leave
  40102d:       c3                      ret

000000000040102e <swap>:
```

重定位表

```
└─$ objdump -r a.o

a.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000016 R_X86_64_PC32     shared-0x0000000000000004
0000000000000023 R_X86_64_PLT32    swap-0x0000000000000004
```

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
} Elf32_Rel;
```

r_offset	重定位入口的偏移。对于**可重定位文件**来说，这个值是该重定位入口所要修正的位置的第一个字节相对于段起始的偏移；对于**可执行文件或共享对象文件**来说，这个值是该重定位入口所要修正的位置的第一个字节的虚拟地址。 我们这里只关心可重定位文件的情况，可执行文件或共享对象文件的情况，将在下一章“动态链接”再作分析
r_info	重定位入口的类型和符号。这个成员的低 8 位表示重定位入口的类型，高 24 位表示重定位入口的符号在符号表中的下标。 因为各种处理器的指令格式不一样，所以重定位所修正的指令地址格式也不一样。每种处理器都有自己一套重定位入口的类型。对于**可执行文件和共享目标文件**来说，它们的重定位入口是动态链接类型的，请参考“动态链接”一章

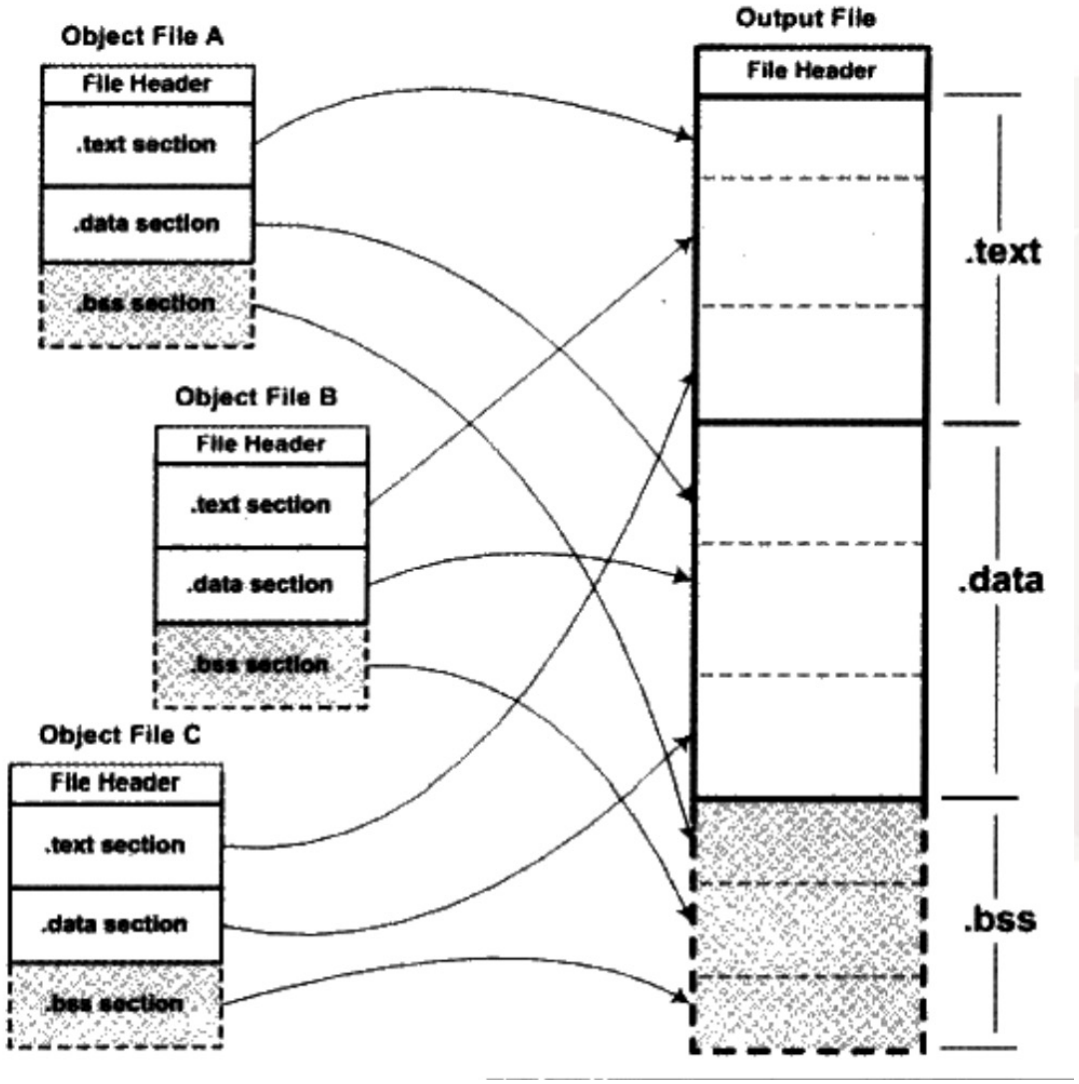
符号解析

```
┌──(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
└─$ ld a.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000000401000
ld: a.o: in function `main':
a.c:(.text+0x16): undefined reference to `shared'
ld: a.c:(.text+0x23): undefined reference to `swap'
```

```
┌──(immortal㉿DESKTOP-9F0PJVS)-[/mnt/…/课程/助教/AAA/link_demo]
└─$ readelf -s a.o

Symbol table '.symtab' contains 6 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS a.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    46 FUNC    GLOBAL DEFAULT    1 main
     4: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND shared
     5: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND swap
```

- 每个目标文件都可能定义了一些符号,也可能引用了定义在其他目标文件的符号.
- 重定位时,链接器会在全局符号表找到相应的符号进行重定位.
- 如果链接器不能再全局符号表中找到这些未定义的符号,那么就会报符号未定义的错误

静态链接



Step 1：空间与地址分配

- 扫描所有的输入目标文件,获得它们的各个段的各种属性
- 收集所有目标文件中的符号定义和符号引用到一个全局符号表
- 合并所有段并建立地址映射关系

Step 2：符号解析和重定位

- 通过step1中收集的信息,读取输入文件中段的数据和重定位信息,进行符号解析和重定位

4 装载

可执行文件

- 静态的概念
- 预先编译好的指令和数据的集合
- 位于磁盘

进程

- 动态的概念
- 程序运行时的一个过程
- 位于内存

每个程序被运行起来以后,它将拥有自己的独立**虚拟地址空间(Virtual Addres Space).**

32位机器的虚拟地址空间为0x00000000~0xFFFFFFFF.
这也是为什么32位机器下指针的大小是4字节.

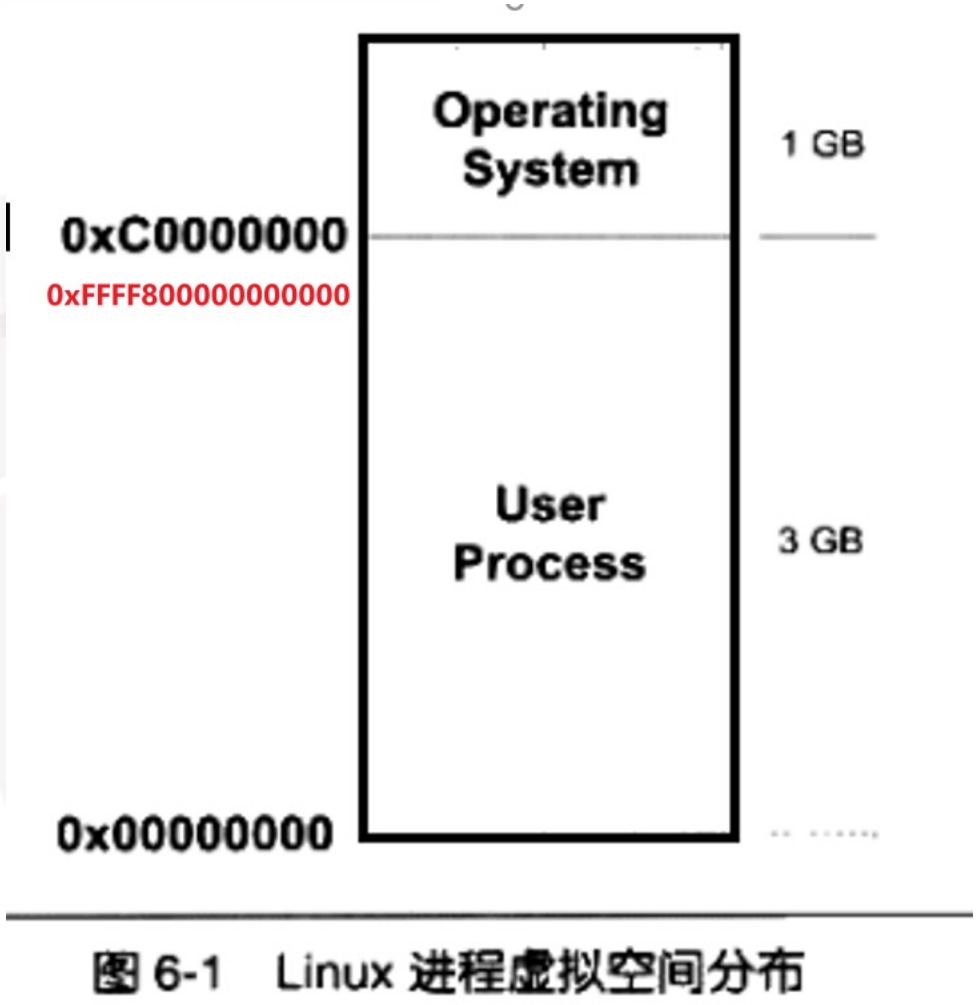


图 6-1 Linux 进程虚拟空间分布

装载方式

- 全部装入
  内存的昂贵
  程序所需要内存有可能大于物理内存

- 动态装载
  利用程序运行时的局部性.程序用到哪个模块就载入该模块到内存,否则就存放在磁盘中
  - Overlay(现在几乎被淘汰)
    由程序员手工控制程序的装载和卸载

将内存和所有磁盘中的数据和指令按照"页"为单位进行划分，之后装载和操作的单位就是页.(目前硬件规定的页的大小有4096字节,8192字节,2MB,4MB等)

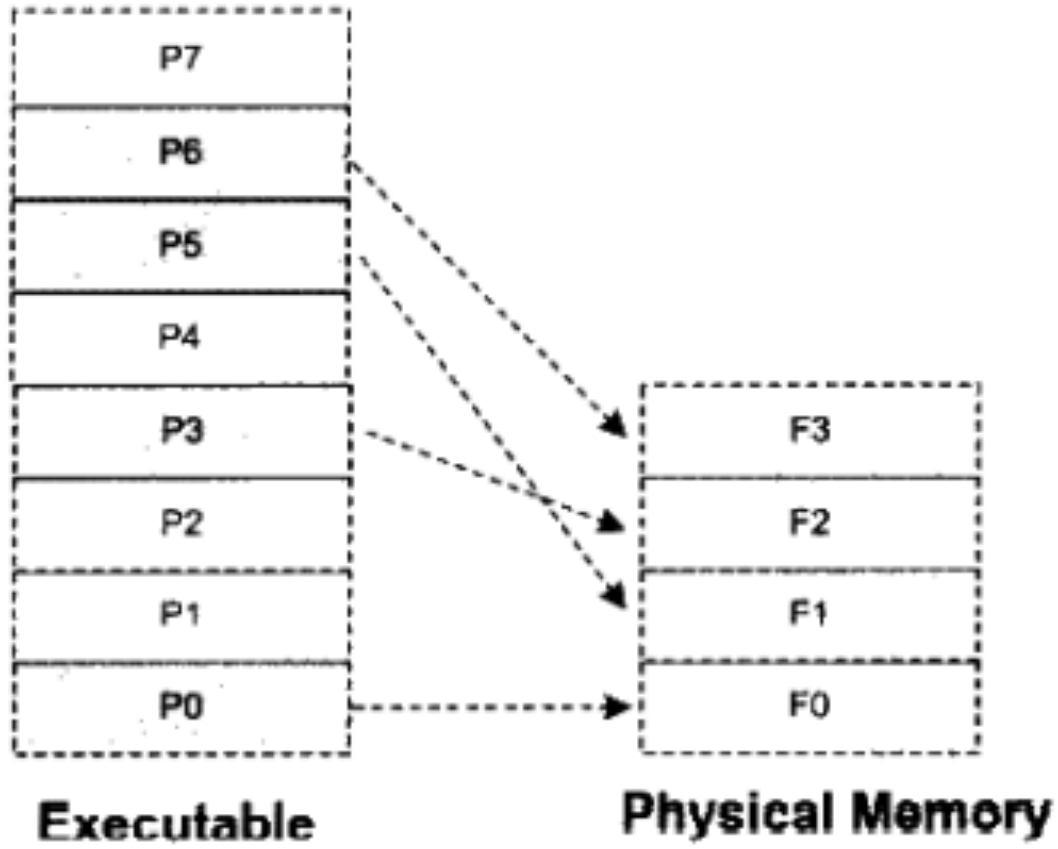


图 6-4 页映射与页装载

进程的建立

- 创建一个独立的虚拟地址空间
- 读取elf文件头,建立虚拟空间和可执行文件的映射关系
- 将CPU指令寄存器设置成可执行文件的入口地址(位于文件头)
- 页错误

- 创建一个独立的虚拟地址空间

前面paging讲到了虚拟空间由一些页映射函数映射至物理空间，那么创建一个虚拟空间实际上并非创建空间而是创建映射函数所需要的数据结构

在i386的linux下，创建虚拟地址空间实际上只是分配了一个数据结构(页目录)

- 读取elf文件头,建立虚拟空间和可执行文件的映射关系

以右图为例,操作系统创建进程以后会在进程相应的数据结构中设置一个.text段的VMA. 它包含的主要信息有:

- 在虚拟空间中的地址(0x08048000- 0x08049000)
- 在对应ELF文件中的偏移(0)
- 对应的属性(读/执行)

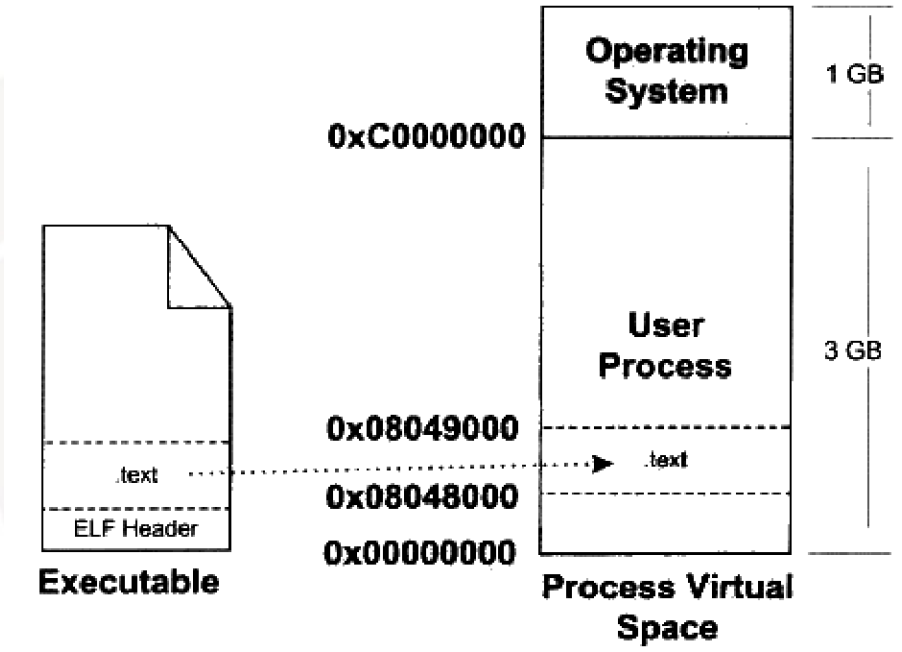


图 6-5 可执行文件与进程虚拟空间

进程的建立

- 将CPU指令寄存器设置成可执行文件的入口地址(位于文件头)

可以理解为操作系统执行了一个跳转指令，跳转到elf文件头中保存的入口地址.

- 页错误

上面三步执行完,可执行文件还没有装载到物理内存中.

当CPU准备执行入口地址时,发现这是一个空页面,这时就会触发页错误.

操作系统查询第二步中建立的数据结构,在物理内存中分配页面

进程重新获得控制权,从页错误发生处重新开始执行

Page的划分

以ELF文件中的段进行划分

- 空间浪费严重

根据段的权限进行划分

- 以代码段为代表的可读可执行段
- 以数据段和bss段为代表的可读可写段
- 以只读数据段代表的只读段

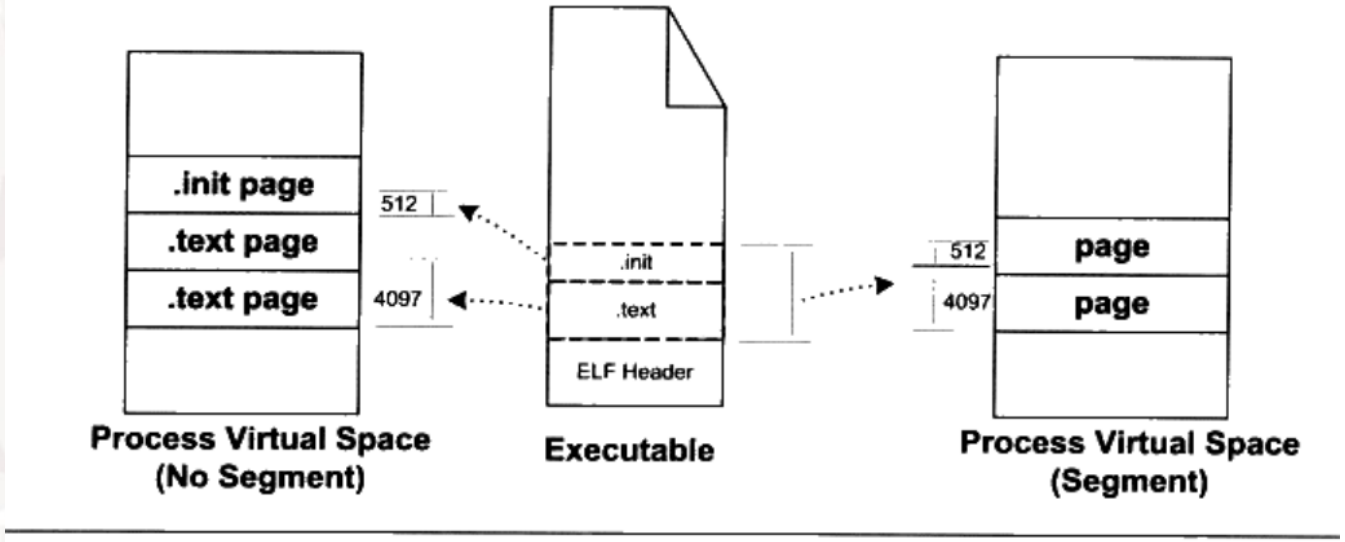
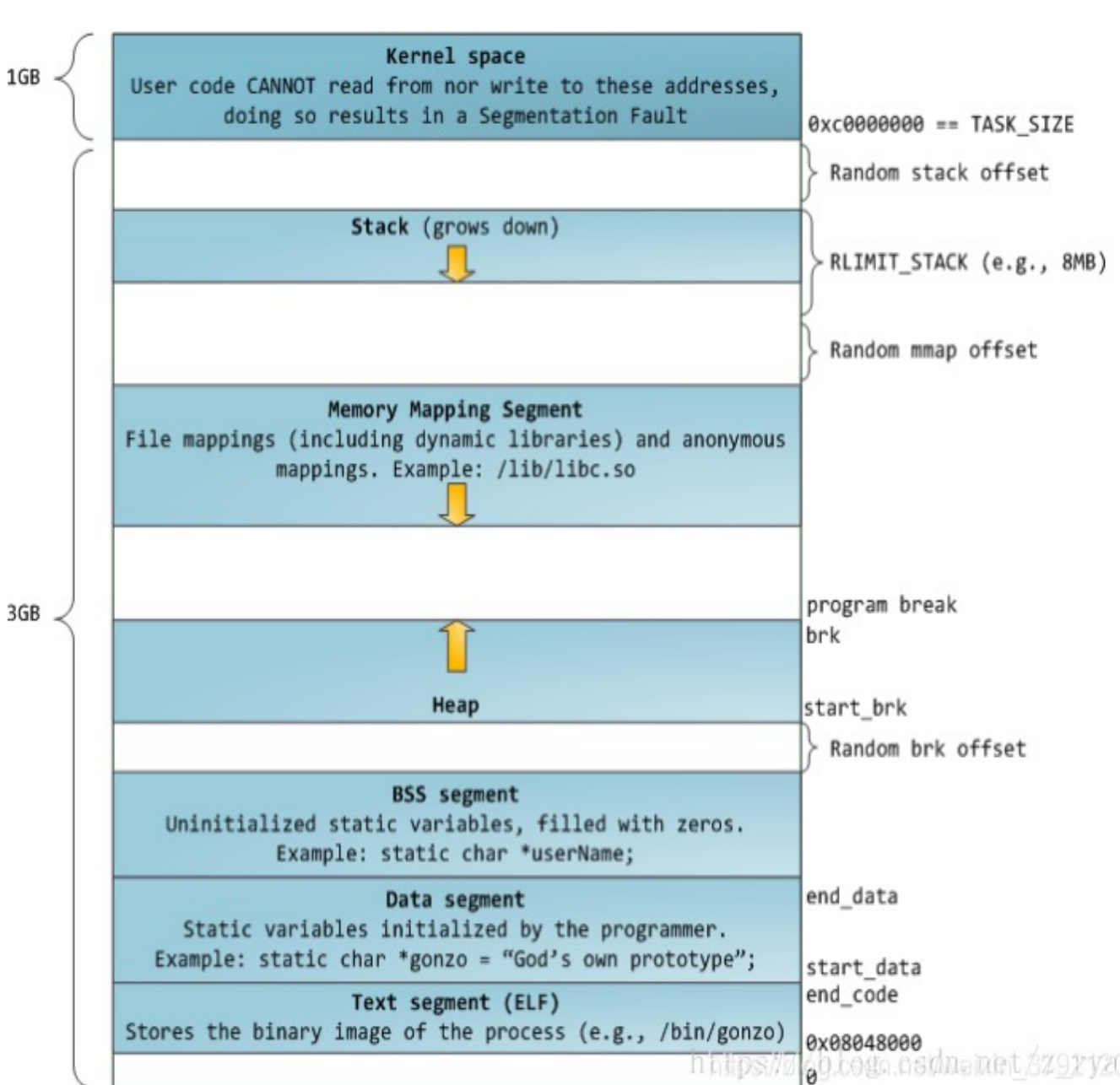


图 6-7 ELF Segment

4

```
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test# cat deadloop.c
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *a = malloc(4);
    while(1){};
    return 0;
}root@iZbp12hfvn1mtwnqhgfpqfZ:~
root@iZbp12hfvn1mtwnqhgfpqfZ:~/
[1] 4519
root@iZbp12hfvn1mtwnqhgfpqfZ:~/
00400000-004b6000 r-xp 00000000
006b6000-006bc000 rw-p 000b6000
006bc000-006bd000 rw-p 00000000
023a0000-023c3000 rw-p 00000000
7ffd587c2000-7ffd587e3000 rw-p
7ffd587f9000-7ffd587fc000 r--p
7ffd587fc000-7ffd587fe000 r-xp
ffffffffff600000-ffffffffff6010
root@iZbp12hfvn1mtwnqhgfpqfZ:~/
```



栈初始化

进程刚开始启动时，需要知道进程运行的环境

- 系统环境变量
- 进程的运行参数

假设我们系统环境变量有"HOME=/home/user"和"PATH=/usr/bin"
运行的两个参数是"prog"和"123"

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

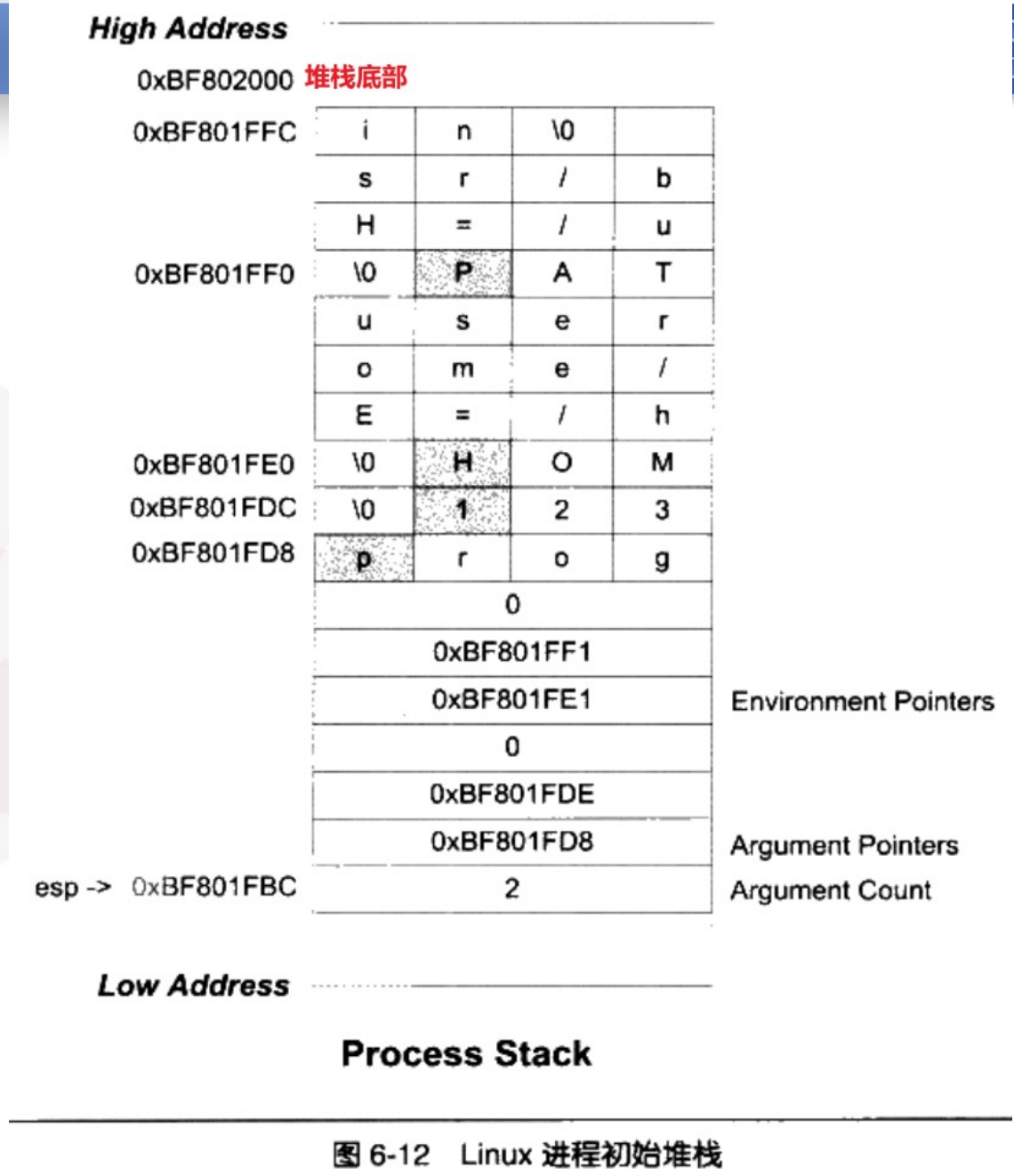


图 6-12 Linux 进程初始堆栈

栈初始化

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

```
25:0128  rsi     0x7fffffffe228 —▸ 0x7fffffffe4a6 ◂— '/root/test/a.out'
26:0130          0x7fffffffe230 ◂— 0x0
27:0138  rdx     0x7fffffffe238 —▸ 0x7fffffffe4b7 ◂— 0x524f4c4f435f534c ('LS_COLOR')
28:0140          0x7fffffffe240 —▸ 0x7fffffffeaa3 ◂— 'SSH_CONNECTION=120.193.7.234 53296 172.17.46.101 22'
29:0148          0x7fffffffe248 —▸ 0x7fffffffead7 ◂— 'LESSCLOSE=/usr/bin/lesspipe %s %s'
2a:0150          0x7fffffffe250 —▸ 0x7fffffffeaf9 ◂— '_=/usr/bin/gdb'
2b:0158          0x7fffffffe258 —▸ 0x7fffffffeb08 ◂— 'LANG=en_US.UTF-8'
2c:0160          0x7fffffffe260 —▸ 0x7fffffffeb19 ◂— 'COLORTERM=truecolor'
2d:0168          0x7fffffffe268 —▸ 0x7fffffffeb2d ◂— 'VSCODE_GIT_ASKPASS_EXTRA_ARGS='
2e:0170          0x7fffffffe270 —▸ 0x7fffffffeb4c ◂— 'S_COLORS=auto'
2f:0178          0x7fffffffe278 —▸ 0x7fffffffeb5a ◂— 'XDG_SESSION_ID=19528'
30:0180          0x7fffffffe280 —▸ 0x7fffffffeb6f ◂— 'USER=root'
31:0188          0x7fffffffe288 —▸ 0x7fffffffeb79 ◂— 'PWD=/root/test'
32:0190          0x7fffffffe290 —▸ 0x7fffffffeb88 ◂— 'LINES=40'
33:0198          0x7fffffffe298 —▸ 0x7fffffffeb91 ◂— 'HOME=/root'
34:01a0          0x7fffffffe2a0 —▸ 0x7fffffffeb9c ◂— 'BROWSER=/root/.vscode-server/bin/c3511e6c69bb39013c4a4b7b9566ec1ca73fc4d5/bin/helpers/browser.sh'
35:01a8          0x7fffffffe2a8 —▸ 0x7fffffffebfd ◂— 'VSCODE_GIT_ASKPASS_NODE=/root/.vscode-server/bin/c3511e6c69bb39013c4a4b7b9566ec1ca73fc4d5/node'
36:01b0          0x7fffffffe2b0 —▸ 0x7fffffffec5c ◂— 'TERM_PROGRAM=vscode'
37:01b8          0x7fffffffe2b8 —▸ 0x7fffffffec70 ◂— 'SSH_CLIENT=120.193.7.234 53296 22'
38:01c0          0x7fffffffe2c0 —▸ 0x7fffffffec92 ◂— 'TERM_PROGRAM_VERSION=1.67.2'
39:01c8          0x7fffffffe2c8 —▸ 0x7fffffffecae ◂— 'VSCODE_IPC_HOOK_CLI=/run/user/0/vscode-ipc-f33d3583-4c8c-470a-9c06-5a519ac978dd.sock'
3a:01d0          0x7fffffffe2d0 —▸ 0x7fffffffed03 ◂— 'COLUMNS=118'
3b:01d8          0x7fffffffe2d8 —▸ 0x7fffffffed0f ◂— 'MAIL=/var/mail/root'
3c:01e0          0x7fffffffe2e0 —▸ 0x7fffffffed23 ◂— 'VSCODE_GIT_ASKPASS_MAIN=/root/.vscode-server/bin/c3511e6c69bb39013c4a4b7b9566ec1ca73fc4d5/extensions/git/dist/a
skpass-main.js'
3d:01e8          0x7fffffffe2e8 —▸ 0x7fffffffeda1 ◂— 'SHELL=/bin/bash'
3e:01f0          0x7fffffffe2f0 —▸ 0x7fffffffedb1 ◂— 'TERM=xterm-256color'
3f:01f8          0x7fffffffe2f8 —▸ 0x7fffffffedc5 ◂— 0x343d4c564c4853 /* 'SHLVL=4' */
40:0200          0x7fffffffe300 —▸ 0x7fffffffedcd ◂— 'LANGUAGE=en_US:en'
41:0208          0x7fffffffe308 —▸ 0x7fffffffeddf ◂— 'VSCODE_GIT_IPC_HANDLE=/run/user/0/vscode-git-0b2e6db6b3.sock'
42:0210          0x7fffffffe310 —▸ 0x7fffffffee1c ◂— 'LOGNAME=root'
43:0218          0x7fffffffe318 —▸ 0x7fffffffee29 ◂— 'GIT_ASKPASS=/root/.vscode-server/bin/c3511e6c69bb39013c4a4b7b9566ec1ca73fc4d5/extensions/git/dist/askpass.sh'
44:0220          0x7fffffffe320 —▸ 0x7fffffffee96 ◂— 'XDG_RUNTIME_DIR=/run/user/0'
45:0228          0x7fffffffe328 —▸ 0x7fffffffeeb2 ◂— 0x6f722f3d48544150 ('PATH=/ro')
46:0230          0x7fffffffe330 —▸ 0x7fffffffefa9 ◂— 'LESSOPEN=| /usr/bin/lesspipe %s'
47:0238          0x7fffffffe338 —▸ 0x7fffffffefc9 ◂— 'LE_WORKING_DIR=/root/.acme.sh'
48:0240          0x7fffffffe340 ◂— 0x0
```

堆

直接使用系统调用

- 系统调用的开销很大，影响性能

运行库提前向操作系统申请一块较大的堆空间，程序需要时从这份空间里取。

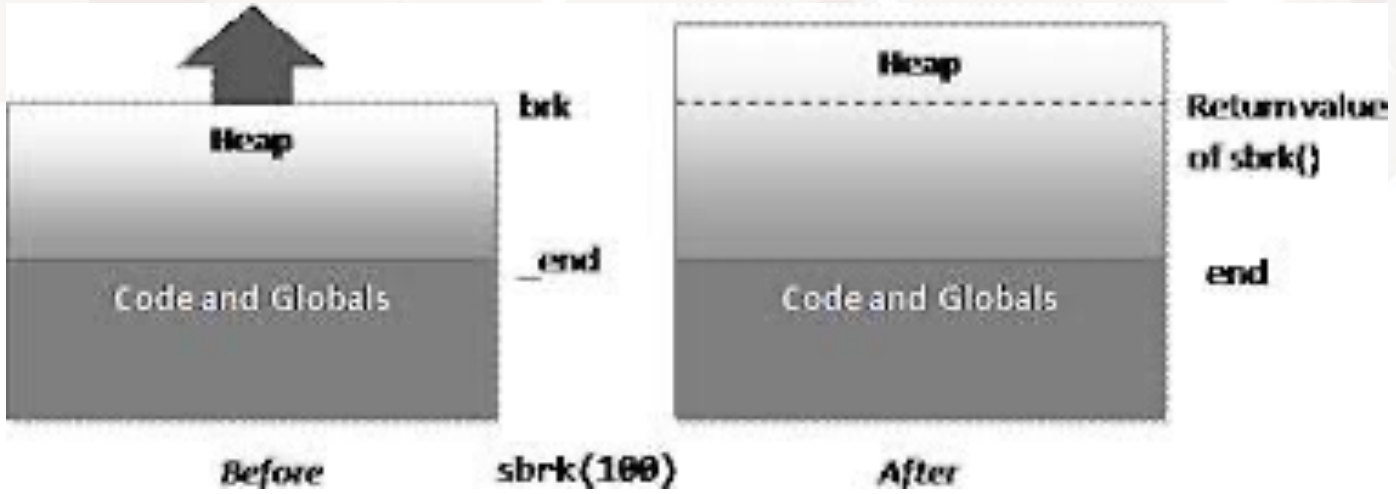
程序的运行库管理堆的分配，glibc下有复杂的堆分配算法

brk

int brk(void* end_data_segment)

- ASLR关闭时，指向数据段的末尾
- ASLR开启时，在数据段末尾加上一段随机的brk offset

brk可以扩大或者缩小数据段，因此扩大的部分可以当做heap段



mmap

void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset);

- 前两个参数指定需要申请空间的起始地址和长度，如果第一个参数设置为0，那么linux系统会自己挑选合适的地址
- prot和flags用于设置申请空间的权限（读，写，执行）和映射类型（文件映射，匿名空间等），最后两个参数是在文件映射时才有用的，我们这里就忽略了

5 动态链接

5 动态链接

静态链接的缺点：

- 极大地浪费了内存空间
- 程序的更新和发布会很麻烦

动态链接：

- 把程序的模块互相分割开来，形成独立的文件
- 把链接这个过程推迟到运行时再进行

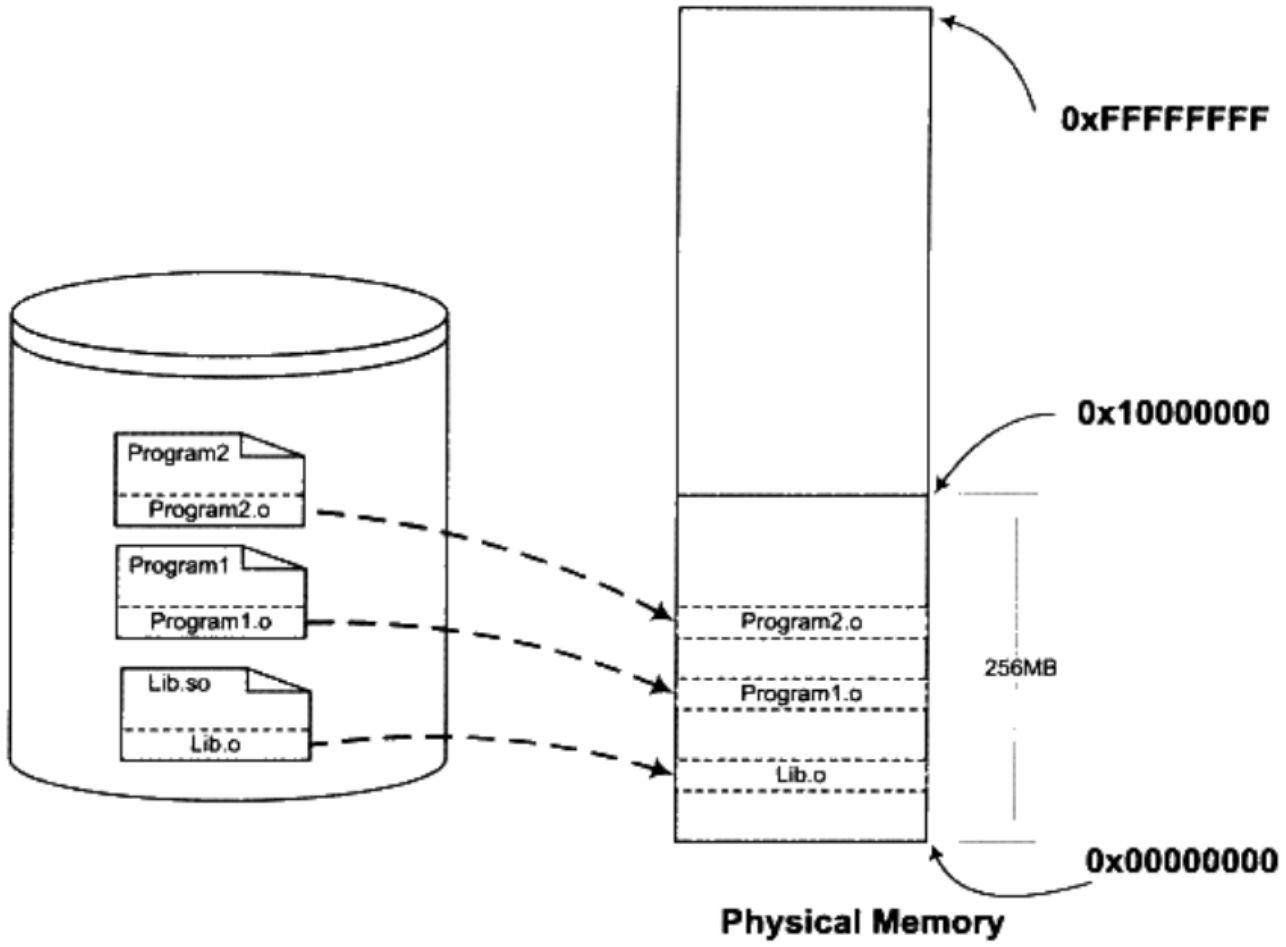


图 7-2　动态链接时文件在内存中的副本

5 动态链接

动态链接设计运行时的链接和多个文件的装载，必须有操作系统的支持

- Linux下，ELF动态链接文件被称为Dynamic Shared Objects，后缀为.so
- Windows下，动态链接文件被称为Dynamic Linking Library，后缀为.dll

动态链接把链接过程推迟到装载的时候，每次装载都需要链接，是否影响效率？

是。但是动态链接的过程可以优化，比如之后会介绍的延迟绑定。据估算，动态链接与静态相比性能损失在5%以下。

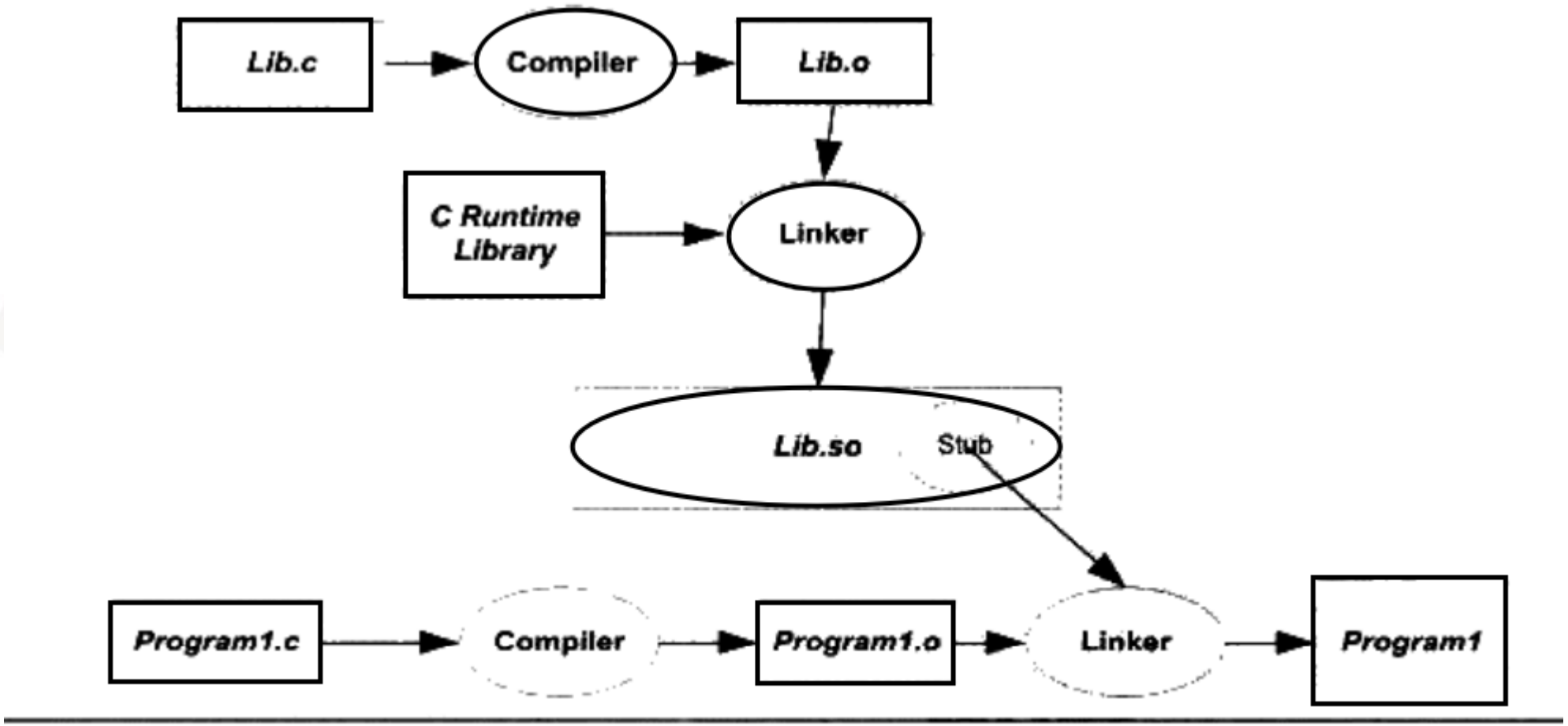

Lib.c
Compiler
Lib.o
C Runtime Library
Linker
Lib.so
Stub
Program1.c
Compiler
Program1.o
Linker
Program1

图 7-3 动态链接过程

例子

```c
/* program1.c */
#include "Lib.h"
int main(){
    foobar(1);
    return 0;
}
```

```c
/* program2.c */
#include "Lib.h"
int main(){
    foobar(2);
    return 0;
}
```

```c
/* Lib.h*/
#ifndef LIB_H
#define LIB_H
void foobar(int i);
#endif
```

```c
/* Lib.c*/
#include <stdio.h>
void foobar(int i){
    printf("From Lib.so,
print %d\n",i);
    sleep(-1);
}
```

```
gcc -fPIC -shared -o Lib.so Lib.c
gcc -o program1 program1.c ./Lib.so
gcc -o program2 program2.c ./Lib.so
```

例子

动态链接形式的C语言运行库libc-2.27.so
Linux下的动态链接器ld-2.27.so

系统开始运行program1之前会先把控制权交给动态链接器，等它把链接工作完成后，再把控制权还给program1

```
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test/Dlink# ./program1 &
[1] 8708
From Lib.so, print 1
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test/Dlink# cat /proc/8708/maps
55cec12cf000-55cec12d0000 r-xp 00000000 fc:01 935705                     /root/test/Dlink/program1
55cec14cf000-55cec14d0000 r--p 00000000 fc:01 935705                     /root/test/Dlink/program1
55cec14d0000-55cec14d1000 rw-p 00001000 fc:01 935705                     /root/test/Dlink/program1
55cec2543000-55cec2564000 rw-p 00000000 00:00 0                          [heap]
7f7f0968b000-7f7f09872000 r-xp 00000000 fc:01 411217                     /lib/x86_64-linux-gnu/libc-2.27.so
7f7f09872000-7f7f09a72000 ---p 001e7000 fc:01 411217                     /lib/x86_64-linux-gnu/libc-2.27.so
7f7f09a72000-7f7f09a76000 r--p 001e7000 fc:01 411217                     /lib/x86_64-linux-gnu/libc-2.27.so
7f7f09a76000-7f7f09a78000 rw-p 001eb000 fc:01 411217                     /lib/x86_64-linux-gnu/libc-2.27.so
7f7f09a78000-7f7f09a7c000 rw-p 00000000 00:00 0
7f7f09a7c000-7f7f09a7d000 r-xp 00000000 fc:01 935704                     /root/test/Dlink/Lib.so
7f7f09a7d000-7f7f09c7c000 ---p 00001000 fc:01 935704                     /root/test/Dlink/Lib.so
7f7f09c7c000-7f7f09c7d000 r--p 00000000 fc:01 935704                     /root/test/Dlink/Lib.so
7f7f09c7d000-7f7f09c7e000 rw-p 00001000 fc:01 935704                     /root/test/Dlink/Lib.so
7f7f09c7e000-7f7f09ca7000 r-xp 00000000 fc:01 411209                     /lib/x86_64-linux-gnu/ld-2.27.so
7f7f09e95000-7f7f09e98000 rw-p 00000000 00:00 0
7f7f09ea5000-7f7f09ea7000 rw-p 00000000 00:00 0
7f7f09ea7000-7f7f09ea8000 r--p 00029000 fc:01 411209                     /lib/x86_64-linux-gnu/ld-2.27.so
7f7f09ea8000-7f7f09ea9000 rw-p 0002a000 fc:01 411209                     /lib/x86_64-linux-gnu/ld-2.27.so
7f7f09ea9000-7f7f09eaa000 rw-p 00000000 00:00 0
7ffe220f3000-7ffe22114000 rw-p 00000000 00:00 0                          [stack]
7ffe221f6000-7ffe221f9000 r--p 00000000 00:00 0                          [vvar]
7ffe221f9000-7ffe221fb000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test/Dlink#
```

地址冲突

可执行文件一定是第一个被加载的文件，因此它可以选择一个固定的空地址。32位机器中，linux下是0x08040000，windows下是0x0040000.

但是共享对象在编译时不能假设自己在进程虚拟地址空间中的位置。即每次加载时，共享对象的加载地址是不同的。这会产生一个问题，程序模块的指令和数据中可能会包含一些绝对地址引用。尤其在跨模块时，会导致地址引用错误。

装载时重定位

地址无关代码

装载时重定位（rebasing）

用于windows系统以及linux系统下未开启-fPIC编译选项时的共享库中。

操作系统在当前模块装载完成以后，对于所有绝对地址引用的地方进行重定位。因为整个模块是按照一个整体被加载的，所以假设原本模块的基地址是0x100,但最后分到的是0x1000，因为模块内指令和数据相对位置没有改变，只需要对每个重定位项加上0x1000-0x100即可。

地址无关代码

1. 模块内部的函数调用、跳转
2. 模块内部的数据访问，比如模块中定义的全局变量、静态变量
3. 模块外部的函数调用、跳转
4. 模块外部的数据访问，比如其他模块中定义的全局变量

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

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

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

Type 2, Inner-module data access

Type 4, Inter-module data access

Type 1, Inner-module call

Type 3, Inter-module call

图 7-4　4 种寻址模式

模块内部的函数调用、跳转

```
8048344 <bar>:
8048344:      55                    push %ebp
8048345:      89 e5                 mov %esp, %ebp
8048347:      5d                    pop %ebp
8048348:      c3                    ret
8048349<foo>:
…
8048357:      e8 e8 ff ff ff        call 8048344 <bar>
804835c:      b8 00 00 00 00        mov $0x0, %eax
…
```

0xfffffffe8 -> -24

0x804835c – 24 = 0x8048344

模块内部的数据访问

```
#include <stdio.h>
#include <stdlib.h>
const char a[] = "hello\n";
int main(){
    printf(a);
    return 0;
}
```

```
000000000000063a <main>:
 63a:   55                      push   %rbp
 63b:   48 89 e5                mov    %rsp,%rbp
 63e:   48 8d 3d a6 00 00 00    lea    0xa6(%rip),%rdi
 645:   e8 c6 fe ff ff          callq  510 <puts@plt>
 64a:   b8 00 00 00 00          mov    $0x0,%eax
 64f:   5d                      pop    %rbp
 650:   c3                      retq
```

0x645+0xa6 = 0x6eb

```
root@iZbp12hfvn1mtwnqhgfpqfZ:~/test# python -c "print(open('./a.out','rb').read()[0x6eb:0x6eb+6])"
hello
```

模块间的数据访问

其他模块的全局变量地址是跟模块装载地址有关的。

ELF的做法是在数据段里建立一个指向这些变量的指针数组，它们被称为全局偏移表（GOT表）。

链接器在装载模块时会查找每个变量所在的地址，然后填充GOT表。

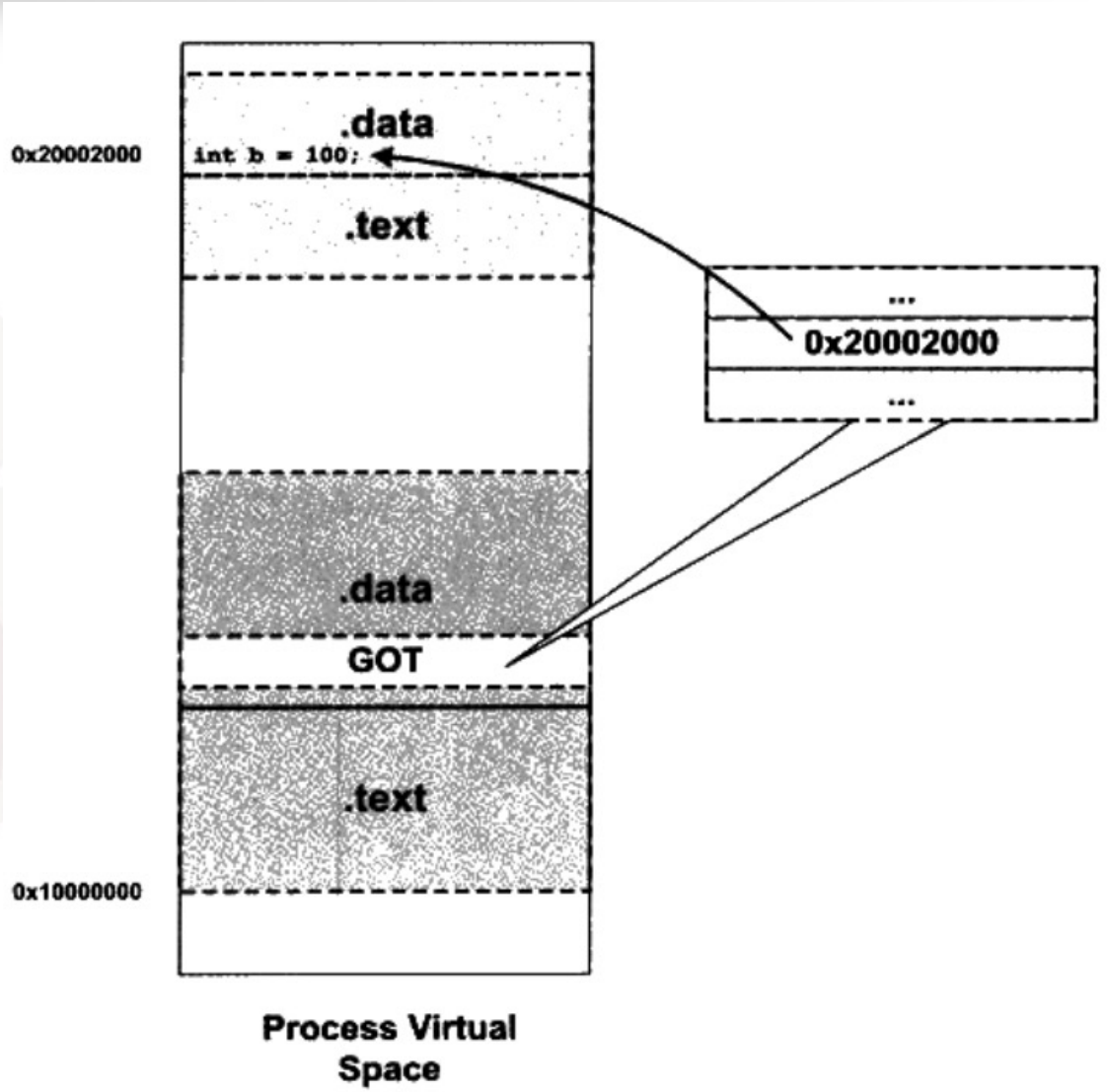


图 7-7　模块间数据访问

模块间的函数调用、跳转

其他模块的函数地址也是跟模块装载地址有关的。

和数据间的访问一样， 需要使用GOT表把函数间的跳转先转换成函数内的跳转。

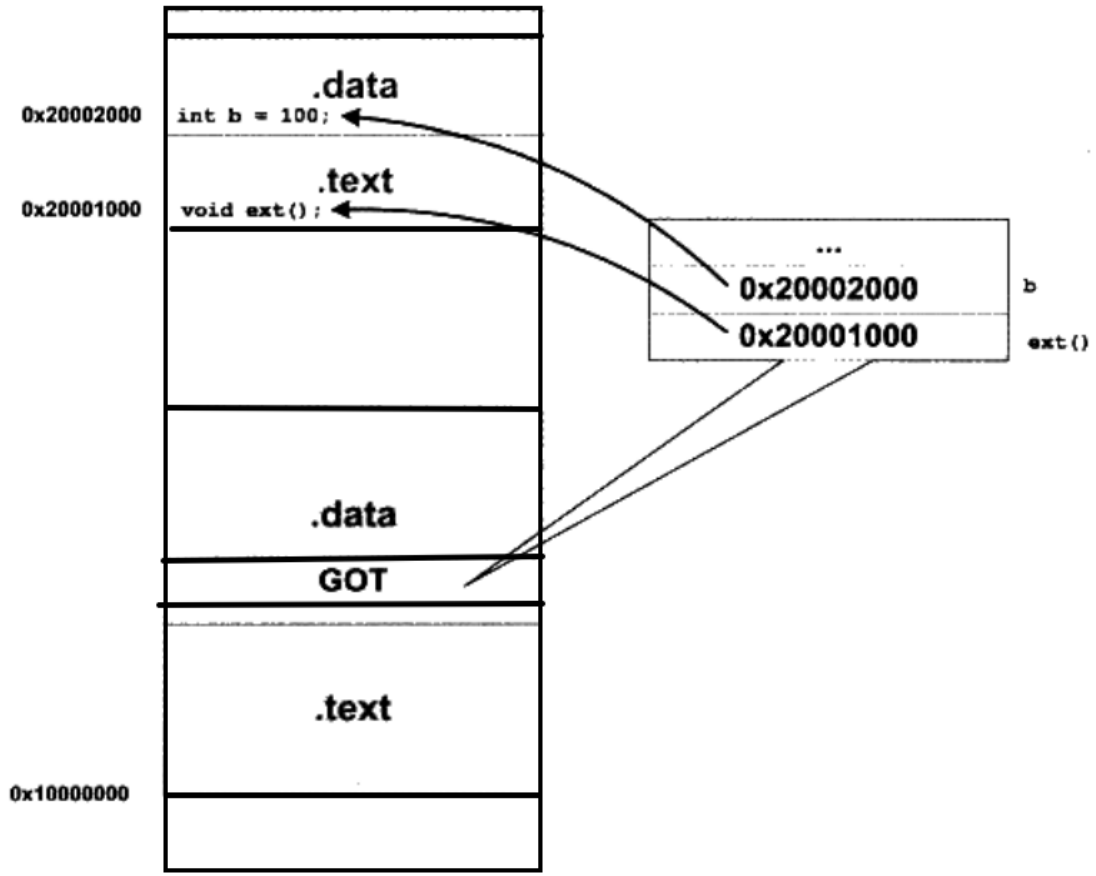


图 7-8 模块间调用、跳转

Lazy Binding（PLT）

动态链接下，模块间存在大量的函数引用，导致动态链接时间较长。ElF采用了延迟绑定的做法，基本思想就是函数第一次被用到时才进行绑定。

动态链接器怎么完成地址绑定工作？需要通过符号知道地址绑定发生的模块和需要绑定的函数。这一步的性能开销是比较大的

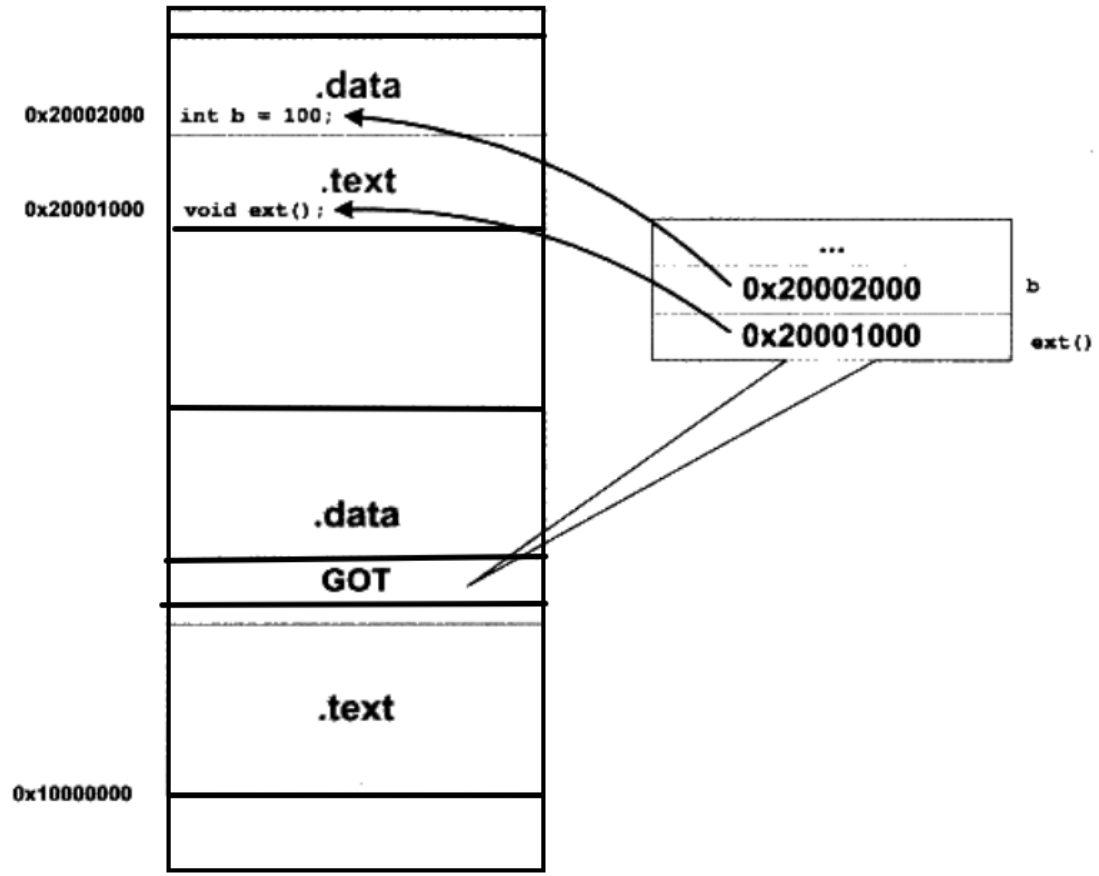


图 7-8　模块间调用、跳转

Lazy Binding（PLT）

PLT为了实现延迟绑定，调用函数并不直接通过GOT表，而是通过一个叫PLT的表项进行跳转

```
bar@plt:
jmp *(bar@GOT)
push n
push moduleID
jmp _dl_runtime_resolve
```

- 初始化前，加粗部分指令实际跳转的位置是push n， 也就是*(bar@GOT) = & (push n)
- 第二条指令将bar这个符号在重定位表“.rel.plt”中的下标压入堆栈
- 第三条指令将bar函数对应的模块id压入堆栈
- 最后调用_dl_runtime_resolve函数，进行一系列工作后，将bar函数真正的地址填入到**bar@GOT**中

数据段地址无关性

```
Static int a;
Static int * p = &a;
```

变量a的地址会随着共享对象的装载地址改变而改变，那么p指针指向的绝对地址也会因此而改变。

选择装载时重定位的方法来解决。
如果数据段中有绝对地址引用，那么编译器和链接器就会产生一个重定位表，这个表里包含了“R_386_RELATIVE”类型的重定位入口，用于解决上述问题。

6 From _start

程序从main开始吗？main运行结束了程序就结束了吗？

```
/* c++ */
#include <string>
using namespace std;
string v;
double foo(){
return 1.0;
}
double g =foo();
int main(){}
```

```
/* c */
#include<stdio.h>
void foo(){
printf("bye!\n");
}
int main(){
atexit(&foo); //atexit注册main之后运行的函数
printf("end of main!\n");
}
```

6 _start

```
public start
start proc near
; __unwind {
xor     ebp, ebp    ; ebp保存上一个函数栈帧，这里取0，表明这就是最外层函数
mov     r9, rdx         ; rtld_fini
pop     rsi             ; argc
mov     rdx, rsp        ; ubp_av
and     rsp, 0FFFFFFFFFFFFFFF0h
push    rax
push    rsp             ; stack_end
mov     r8, offset __libc_csu_fini ; fini
mov     rcx, offset __libc_csu_init ; init
mov     rdi, offset main ; main
addr32 call    __libc_start_main
hlt
; } // starts at 4016A0
start endp
```

```
int __libc_start_main(
int *(main) (int, char * *, char * *),
int argc,
char * * ubp_av,
void (*init) (void),
void (*fini) (void),
void (*rtld_fini) (void),
void (* stack_end)
);
```

X64传参顺序： rdi, rsi, rdx, rcx, r8, r9。 之后还有多的参数则和x32一致，用栈传参。

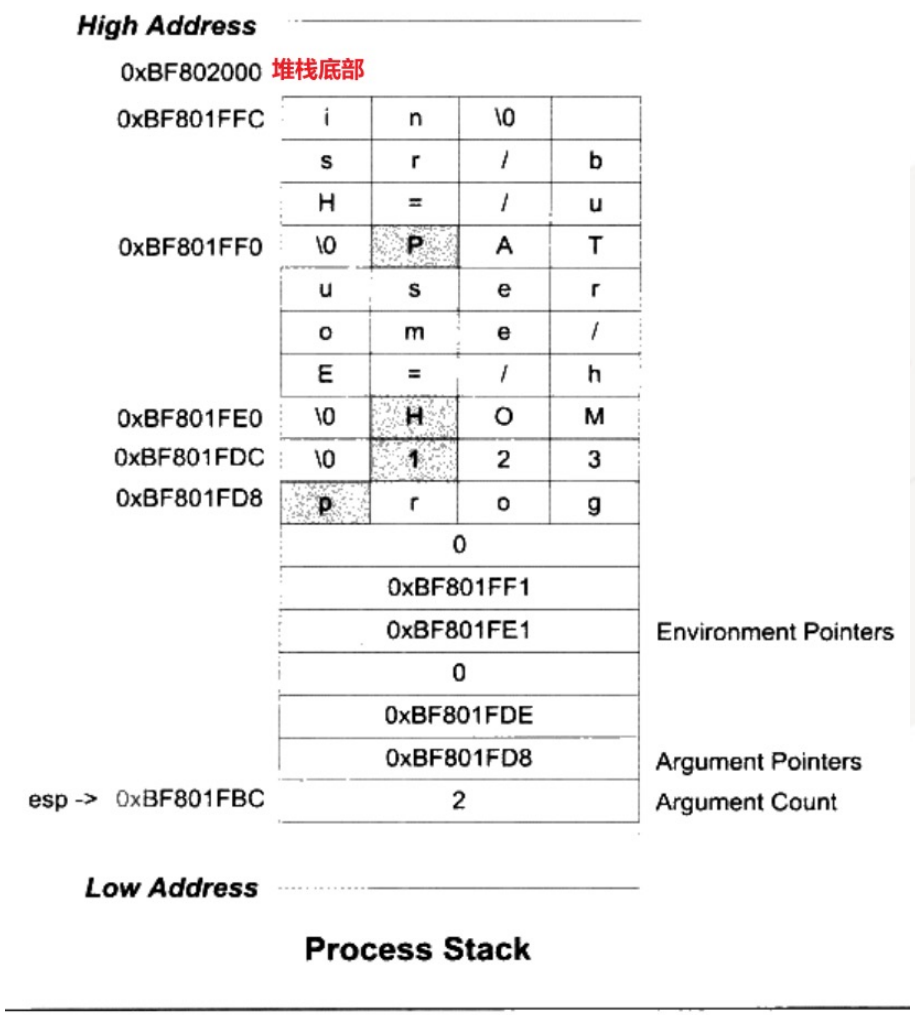


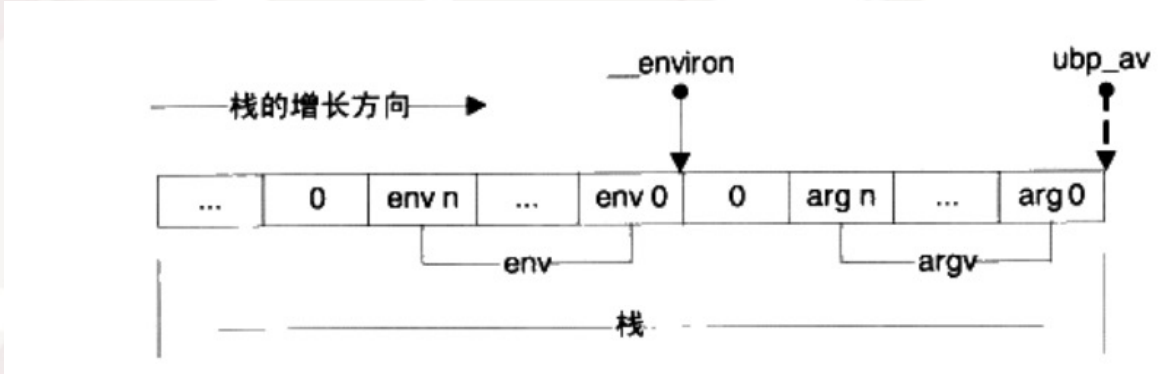
图 6-12 Linux 进程初始堆栈

```
public start
start proc near
; __unwind {
xor     ebp, ebp
mov     r9, rdx         ; rtld_fini
pop     rsi             ; argc
mov     rdx, rsp        ; ubp_av
and     rsp, 0FFFFFFFFFFFFFFF0h
push    rax
push    rsp             ; stack_end
mov     r8, offset __libc_csu_fini ; fini
mov     rcx, offset __libc_csu_init ; init
mov     rdi, offset main ; main
addr32 call    __libc_start_main
hlt
; } // starts at 4016A0
start endp
```

6 __libc_start_main

```
int __libc_start_main(
int *(main) (int, char * *, char * *),
int argc,
char * * ubp_av,
void (*init) (void),
void (*fini) (void),
void (*rtld_fini) (void),
void (* stack_end)
);
```

```
char ** ubp_ev=&ubp_av[argc+1];
__environ = ubp_ev;
__libc_stack_end = stack_end;
```



_start -> __libc_start_main -> exit

__pthread_initialize_minimal(); //初始化线程相关的功能
__cxa_atexit(rtld_fini,NULL,NULL);//注册退出函数
__Libc_init_first(argc, argv, __environ);//初始化libc
__cxa_atexit(fini,null,null)
(*init)(argc, argv, __environ);//通过传入的指针（init）调用程序初始化函数。
result = main(argc, argv, __environ);
exit(result);

6 __libc_start_main

```
void __fastcall _libc_csu_init(unsigned int a1, __int64 a2, __int64 a3)
{
  signed __int64 v4; // r14
  __int64 i; // rbx

  init_proc();
  v4 = _do_global_dtors_aux_fini_array_entry - _frame_dummy_init_array_entry;
  if ( v4 )
  {
    for ( i = 0LL; i != v4; ++i )
      ((void (__fastcall *)(_QWORD, __int64, __int64))_frame_dummy_init_array_entry[i])(a1, a2, a3);
  }
}
void _libc_csu_fini(void)
{
  signed __int64 i; // rbx

  for ( i = (&_gettext_germanic_plural - (_UNKNOWN *)_do_global_dtors_aux_fini_array_entry) >> 3; i; --i )
    _do_global_dtors_aux_fini_array_entry[i - 1]();
  term_proc();
}
```

```
.init_array:00000000004B0948                    ; ===========================================================================
.init_array:00000000004B0948
.init_array:00000000004B0948                    ; Segment type: Pure data
.init_array:00000000004B0948                    ; Segment permissions: Read/Write
.init_array:00000000004B0948                    _init_array     segment qword public 'DATA' use64
.init_array:00000000004B0948                                    assume cs:_init_array
.init_array:00000000004B0948                                    ;org 4B0948h
.init_array:00000000004B0948 90 17 40 00 00 00 00 00 __frame_dummy_init_array_entry dq offset frame_dummy
.init_array:00000000004B0948                                                        ; DATA XREF: __libc_csu_init+2↑o
.init_array:00000000004B0948                                                        ; __libc_csu_init+B↑o ...
.init_array:00000000004B0948                    _init_array     ends                ; Alternative name is '__init_array_start'
.init_array:00000000004B0948
.fini_array:00000000004B0950                    ; ===========================================================================
.fini_array:00000000004B0950
.fini_array:00000000004B0950                    ; Segment type: Pure data
.fini_array:00000000004B0950                    ; Segment permissions: Read/Write
.fini_array:00000000004B0950                    _fini_array     segment qword public 'DATA' use64
.fini_array:00000000004B0950                                    assume cs:_fini_array
.fini_array:00000000004B0950                                    ;org 4B0950h
.fini_array:00000000004B0950 50 17 40 00 00 00 00 00 __do_global_dtors_aux_fini_array_entry dq offset __do_global_dtors_aux
.fini_array:00000000004B0950                                                        ; DATA XREF: __libc_csu_init+4C↑o
.fini_array:00000000004B0950                                                        ; __libc_csu_fini+1↑o
.fini_array:00000000004B0950                                                        ; Alternative name is '__fini_array_start'
.fini_array:00000000004B0958 60 16 40 00 00 00 00 00                dq offset fini
.fini_array:00000000004B0958                    _fini_array     ends
.fini_array:00000000004B0958
```

_start -> __libc_start_main -> exit -> _exit

```
void exit( int status)
{
        while( __exit_funcs != NULL){
                …
                __exit_funcs = __exit_funcs->next;
        }
        …
        _exit(status);
}
```

6 __libc_start_main

THANK YOU