4-静态链接

程序内部分成.text和.data

- 隔离性：两部分权限可以被分为可读写和只读
- 缓存友好：现代CPU L1缓存设计为指令缓存和数据缓存，更好地利用程序局部性
- 共享指令：多个进程副本共享指令，节省虚拟内存空间

空间和地址分配

- 相似段合并：将所有目标文件（.o文件）的.text，.data，.bss段合并到可执行文件的对应段中
- 链接器为目标文件分配地址和空间
  - 输出的**可执行文件中的空间**
  - **装载**后的虚拟地址中的**虚拟地址空间**

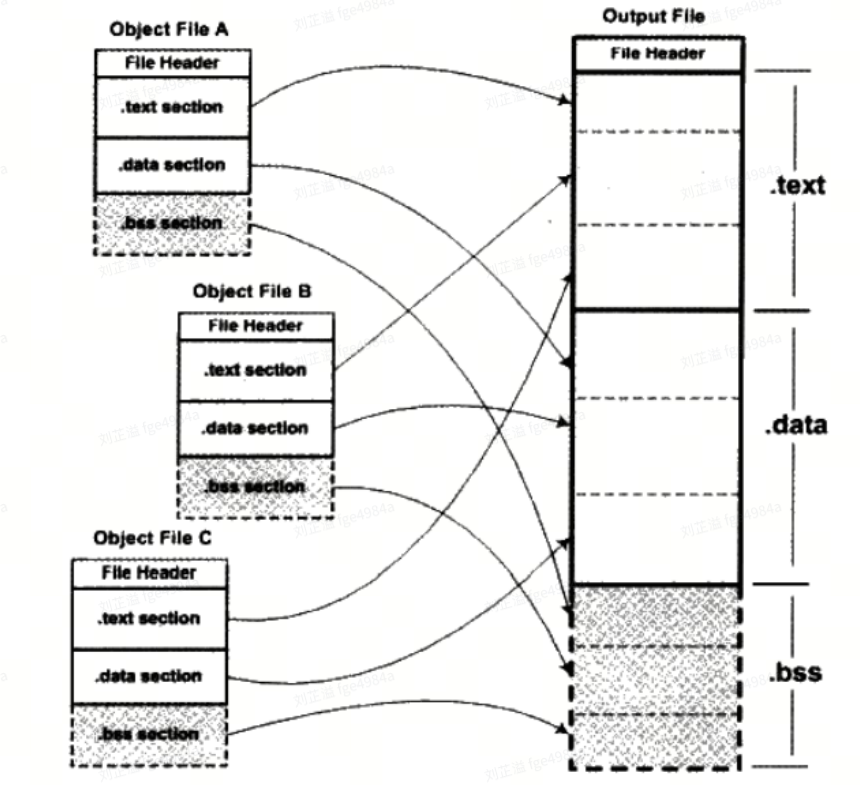

图 4-2　实际的空间分配策略

两步链接

- 空间和地址分配：扫描输入的目标文件，获得段长度，属性和位置；**将符号表中信息收集起来放到全局符号表；将段合并并建立映射关系**
- 符号解析和重定位：读取文件中段的数据、重定位信息；进行**符号解析和重定位**

- 链接前后的程序中的地址已经是程序再进程中的虚拟地址
- 链接前，目标文件的所有段的VMA都是0；**因为虚拟空间还没分配，所以VMA默认为0**；链接后，**各个段被分配到相应的虚拟地址**

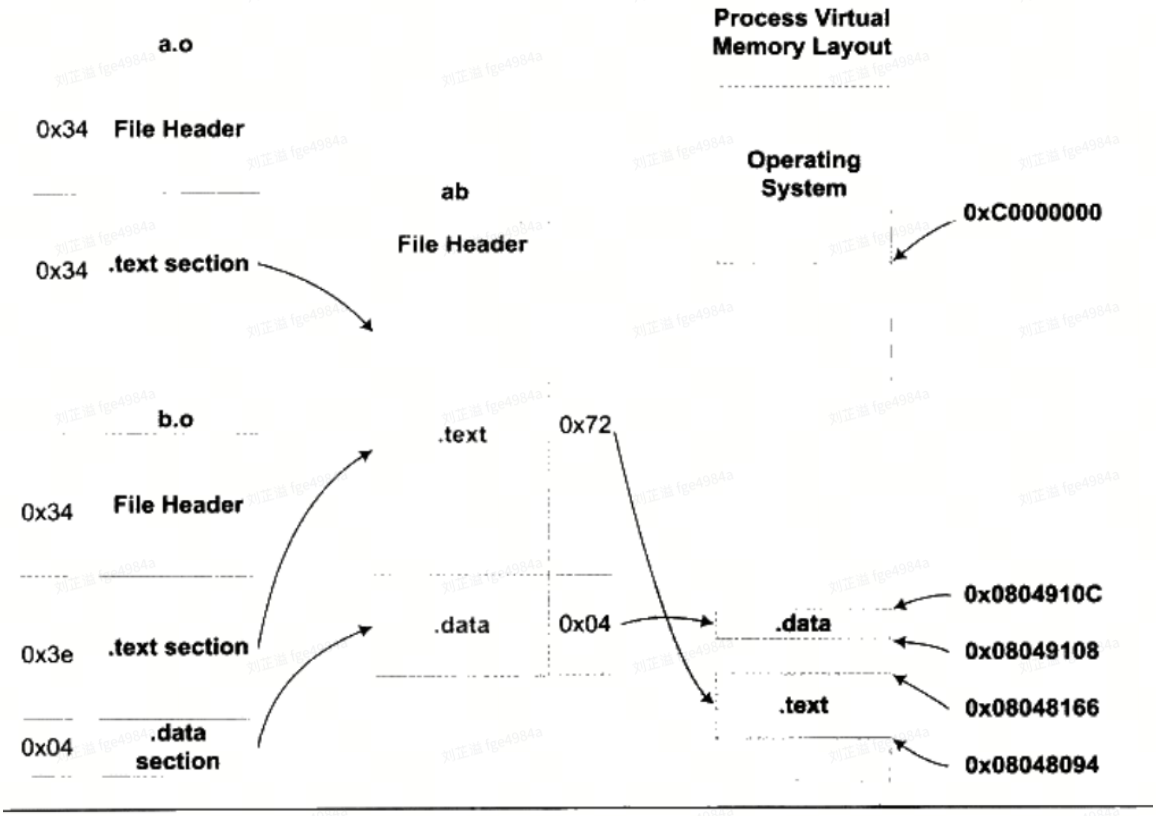

图 4-3　目标文件、可执行文件与进程空间

符号解析和重定位

重定位

```
// a.c
void bar() {
  int bar = 1;
}

_start() {
  foo();
  bar();
}

// b.c
void foo() {
  int a = 1;
}
```

- 链接前，VMA均为0

```
lzy@lzydesktop:~/Workspace/linklib/staticlink$ objdump -h a.o

a.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000031  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000071  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000071  2**0
                  ALLOC
  3 .comment      0000002c  0000000000000000  0000000000000000  00000071  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  0000009d  2**0
                  CONTENTS, READONLY
  5 .note.gnu.property 00000020  0000000000000000  0000000000000000  000000a0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000c0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

```
lzy@lzy:~/Workspace/Linker_Lib/static_linker$ objdump -d a.o

a.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <bar>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%rbp)
   f:   90                      nop
  10:   5d                      pop    %rbp
  11:   c3                      retq

0000000000000012 <_start>:
  12:   f3 0f 1e fa             endbr64
  16:   55                      push   %rbp
  17:   48 89 e5                mov    %rsp,%rbp
  1a:   b8 00 00 00 00          mov    $0x0,%eax
  1f:   e8 00 00 00 00          callq  24 <_start+0x12>
  24:   b8 00 00 00 00          mov    $0x0,%eax
  29:   e8 00 00 00 00          callq  2e <_start+0x1c>
  2e:   90                      nop
  2f:   5d                      pop    %rbp
  30:   c3                      retq
```

```
lzy@lzydesktop:~/Workspace/linklib/staticlink$ objdump -h b.o

b.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000012  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000052  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000052  2**0
                  ALLOC
  3 .comment      0000002c  0000000000000000  0000000000000000  00000052  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  0000007e  2**0
                  CONTENTS, READONLY
  5 .note.gnu.property 00000020  0000000000000000  0000000000000000  00000080  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .eh_frame     00000038  0000000000000000  0000000000000000  000000a0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

```
lzy@lzy:~/Workspace/Linker_Lib/static_linker$ objdump -d b.o

b.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%rbp)
   f:   90                      nop
  10:   5d                      pop    %rbp
  11:   c3                      retq
```

- 链接后的ab文件

```
lzy@lzydesktop:~/Workspace/linklib/staticlink$ objdump -h ab

ab:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .note.gnu.property 00000020  0000000000400190  0000000000400190  00000190  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         00000043  0000000000401000  0000000000401000  00001000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .eh_frame     00000078  0000000000402000  0000000000402000  00002000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .comment      0000002b  0000000000000000  0000000000000000  00002078  2**0
                  CONTENTS, READONLY
```

```
lzy@lzy:~/Workspace/Linker_Lib/static_linker$ objdump -d ab

ab:     file format elf64-x86-64


Disassembly of section .text:

0000000000001000 <bar>:
    1000:       f3 0f 1e fa             endbr64
    1004:       55                      push   %rbp
    1005:       48 89 e5                mov    %rsp,%rbp
    1008:       c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%rbp)
    100f:       90                      nop
    1010:       5d                      pop    %rbp
    1011:       c3                      retq

0000000000001012 <_start>:
    1012:       f3 0f 1e fa             endbr64
    1016:       55                      push   %rbp
    1017:       48 89 e5                mov    %rsp,%rbp
    101a:       b8 00 00 00 00          mov    $0x0,%eax
    101f:       e8 0d 00 00 00          callq  1031 <foo>
    1024:       b8 00 00 00 00          mov    $0x0,%eax
    1029:       e8 d2 ff ff ff          callq  1000 <bar>
    102e:       90                      nop
    102f:       5d                      pop    %rbp
    1030:       c3                      retq

0000000000001031 <foo>:
    1031:       f3 0f 1e fa             endbr64
    1035:       55                      push   %rbp
    1036:       48 89 e5                mov    %rsp,%rbp
    1039:       c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%rbp)
    1040:       90                      nop
    1041:       5d                      pop    %rbp
    1042:       c3                      retq
```

- 使用readelf查看符号表，在a.o中foo是未定义的符号

```
lzy@lzy:~/Workspace/Linker_Lib/static_linker$ readelf -s a.o

Symbol table '.symtab' contains 13 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
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     9: 0000000000000000    18 FUNC    GLOBAL DEFAULT    1 bar
    10: 0000000000000012    31 FUNC    GLOBAL DEFAULT    1 _start
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TA
BLE_
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo
```

指令修正方式

- readelf -r a.o查看重定位表

```
lzy@lzy:~/Workspace/Linker_Lib/static_linker$ readelf -r a.o

Relocation section '.rela.text' at offset 0x280 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Ad
dend
000000000020  000c00000004 R_X86_64_PLT32    0000000000000000 foo - 4
00000000002a  000900000004 R_X86_64_PLT32    0000000000000000 bar - 4

Relocation section '.rela.eh_frame' at offset 0x2b0 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Ad
dend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
000000000040  000200000002 R_X86_64_PC32     0000000000000000 .text + 12
```

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

- S是符号的实际地址：这里foo是0x1031
- A是被修正位置的值，call指令后面的值：这里是0x0
- P是被修正的位置；是虚拟地址（段基址+r_offset）：这里是0x1000 + 0x20 + 4

绝对地址修正

- S+A

相对地址修正

- S+A-P，这里是0x0d

COMMON块

- 处理多个弱符号时使用COMMON块机制；**取最大的符号分配空间**
- 如果有弱符号空间大于强符号空间；ld会报警告
- 链接前，未初始化的全局变量放在COMMON块中，防止在其他文件中有更大的弱符号；**链接后大小确定，可以将全局变量放到.bss段**
- 不使用COMMON块处理未初始化的变量

- gcc的“-fno-common”

```
int global __attribute__((nocommon));
// 如果有其他文件初始化同样的变量会保错
```

C++相关问题

重复代码消除

- 重复代码包括模板实例化；外部内联函数以及虚函数表；造成如下问题
  - 空间浪费
  - 地址容易出错
  - 指令运行效率较低；一份指令多份副本，指令Cache命中率较低
- **如果模板实例化，在目标文件中加入实例化的段；如果其他编译单元也实例化该段，直接合并最后的代码段**
- **函数级别链接**
  - 将所有函数单独保存在一个段中，链接器使用的函数合并到输出文件，其余函数直接丢弃
  - "-ffunction-sections"和"-fdata-sections"分别将函数和数据保存到独立的段中

全局构造和析构

- .init：保存的是可执行指令；进程初始化代码
- .fini：保存进程终止指令代码
- C++全局构造在.init段；全局析构在.fini段

静态库链接

- 查看glibc的目标文件

gcc编译过程

- --verbose可以查看编译过程
- **cc1是C语言编译器；as是GNU的汇编器；collect2可以看作是ld链接器的包装，后面的库被链接到目标文件中**

```
 /usr/lib/gcc/x86_64-linux-gnu/9/cc1 -quiet -v -imultiarch x86_64-linux-gnu hello.c -quiet -dumpbase hello.c -mtune=generic -march=x86-64 -auxbase hello -version -fno-builtin -fasynchronous-unwind-tables -fstack-protector-strong -Wformat -Wformat-security -fstack-clash-protection -fcf-protection -o /tmp/ccdVN6qz.s
GNU C17 (Ubuntu 9.4.0-1ubuntu1~20.04.2) version 9.4.0 (x86_64-linux-gnu)
        compiled by GNU C version 9.4.0, GMP version 6.2.0, MPFR version 4.0.2, MPC version 1.1.0, isl version isl-0.22.1-GMP

as -v --64 -o /tmp/ccpCgjUz.o /tmp/ccdVN6qz.s

/usr/lib/gcc/x86_64-linux-gnu/9/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper -plugin-opt=-fresolution=/tmp/cczH3diC.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_eh -plugin-opt=-pass-through=-lc --build-
```

```
id -m elf_x86_64 --hash-style=gnu --as-needed -static -z relro
/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/9/crtbeginT.o -L/usr/lib/gcc/x86_64-linux-gnu/9
-L/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu -
L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib -L/lib/x86_64-linux-gnu -
L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -
L/usr/lib/gcc/x86_64-linux-gnu/9/../../.. /tmp/ccpCgjUz.o --start-group -lgcc
-lgcc_eh -lc --end-group /usr/lib/gcc/x86_64-linux-gnu/9/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crtn.o
```

链接过程控制

三种方法

- 使用命令行指定链接器参数
- 将链接指令存放在目标文件中
- 链接控制脚本：使用-T来指定运行链接脚本 ld -T link.script

hello.c的链接

```c
char *str = "hello, world\n";
void myprint(){
  asm(
    "movq $1, %%rax \n"   // write syscall index
    "movq $1, %%rdi \n"   // stdout
    "movq %0, %%rsi \n"   // string address
    "movq $13, %%rdx \n"  // length of string
    "syscall \n"
    : // no output
    : "r"(str) : "rax", "rdi", "rsi", "rdx");
}
void myexit(){
  asm(
    "movq $60, %rax \n"   // exit syscall index
    "xor %rdi, %rdi \n"   // exit code 0
    "syscall \n");
}
int nomain() {
  myprint();
  myexit();
}
```

```
ENTRY(nomain)
SECTIONS
{
  . = 0x08048000 + SIZEOF_HEADERS;
  tinytext : {
    *(.text)
    *(.data)
    *(.rodata)
  }
  /DISCARD/ : {
    *(.eh_frame)
    *(.comment)
  }
}
```

- Gcc -c -fno-builtin hello.c
- Ld -static -T hello.ld -o hello hello.o

```
lzy@lzydesktop:~/Workspace/linklib/staticlink$ readelf -S hello
There are 7 section headers, starting at offset 0x2b8:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .note.gnu.pr[...] NOTE            0000000008048120  00000120
       0000000000000020  0000000000000000   A       0     0     8
  [ 2] tinytext          PROGBITS        0000000008048140  00000140
       0000000000000070  0000000000000000 WAX       0     0     1
  [ 3] .data.rel.local   PROGBITS        00000000080481b0  000001b0
       0000000000000008  0000000000000000  WA       0     0     8
  [ 4] .symtab           SYMTAB          0000000000000000  000001b8
       0000000000000090  0000000000000018           5     2     8
  [ 5] .strtab           STRTAB          0000000000000000  00000248
       0000000000000023  0000000000000000           0     0     1
  [ 6] .shstrtab         STRTAB          0000000000000000  0000026b
       0000000000000047  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

- 常用的命令语句

表 4-4

命令语句	说明
ENTRY(*symbol*)	指定符号 *symbol* 的值为入口地址（Entry Point）。入口地址即进程执行的第一条用户空间的指令在进程地址空间的地址，它被指定在 ELF 文件头 Elf32_Ehdr 的 e_entry 成员中。ld 有多种方法可以设置进程入口地址，它们之间的优先级按以下顺序排列（编号越靠前，优先级越高）： 1. ld 命令行的-e 选项 2. 链接脚本的 ENTRY(symbol)命令 3. 如果定义了_start 符号，使用_start 符号值 4. 如果存在.text 段，使用.text 段的第一字节的地址 5. 使用值 0
STARTUP(*filename*)	将文件 filename 作为链接过程中的第一个输入文件。具体请参见“链接顺序”
SEARCH_DIR(*path*)	将路径 path 加入到 ld 链接器的库查找目录。ld 会根据指定的目录去查找相应的库。跟 “-L*path*” 命令有着相同的作用

续表

命令语句	说明
INPUT(*file, file, ...*) INPUT(*file file ...*)	将指定文件作为链接过程中的输入文件
INCLUDE *filename*	将指定文件包含进本链接脚本。类似于 C 语言中的#include 预处理
PROVIDE(*symbol*)	在链接脚本中定义某个符号。该符号可以在程序中被引用。其实前文提到的特殊符号都是由系统默认的链接脚本通过 PROVIDE 命令定义在脚本中的