操作系统实验（二）问答题参考

南京大学软件学院

2015.5

实验重点

本次作业重点：熟悉掌握 Fat12 文件系统，$gcc + nasm$ 联合编译实践以及了解实模式与保护模式的基本内容。

1 问题清单

在整个实验的过程中，无论是编程还是查资料，请各位同学注意思考以下问题，助教检查时会从中随机抽取数个题目进行提问，根据现场作答给出分数。请注意，我们鼓励自己思考和动手实验，如果能够提供自己的思考结果并辅助以相应的实验结果进行说明，在分数评定上会酌情考虑。

1.1 PPT 相关内容

1. 实模式下的寻址方式以及实模式的缺陷

   寻址方式：实模式下，段在内存中固定的位置（物理地址 = 段值 *16 + 偏移）；

   缺陷：通过改变段寄存器的值，我们可以随心所欲的访问内存任何一个单元，而丝毫不受到限制，不能对内存访问加以限制，也就谈不上对系统的保护；内存中每个字节的地址能由不止一个的段基址加偏移表示，比如 04808 能够由 047C:0048,047D:0038,047E:0028 or 047B:0058 表示，分段地址之间的比较将复杂化。

2. 保护模式下的寻址过程：

   - 段寄存器中存储的是什么?GDT 是什么?LDT 是什么? 如何区分 LDT 和 GDT? LDT 和 GDT 的区别是什么? 如何定位到 Descriptor?Descriptor 的内容有哪些?
     - 存储的是选择子，即描述符在描述符表中的位置
     - GDT 是全局描述符表；LDT 是局部描述符表

  - 根据选择子中的第三位决定是 GDT 还是 LDT
  - DT(Local) 与 GDT 相同，但是不是全局的，对于某个进程，它只知道它自己的 LDT。每个进程有自己的 LDT，访问自己的段时从 LDT 查询。进程从 LDTR 寄存器中获得 LDT 的位置，向它发起查询
  - 如果是 GDT，则根据 GDTR 和段寄存器中的内容定位到描述符；如果是 LDT，则根据 LDTR 中的内容（选择子）和 GDTR 中的内容定位到描述符
  - Descriptor 中的内容包括是否在内存中，段的起始地址、界限、属性等内容
- GDTR 中的内容是什么?LDTR 中存储的是什么? 为什么 LDT 要放在 GDT 中?
  - 全局描述符表的位置
  - 选择子，用于定位 GDT 中的某个 LDT 描述符，得到 LDT 的地址
  - GDT 表只有一个，是固定的；而 LDT 表每个任务就可以有一个，因此有多个，并且由于任务的个数在不断变化其数量也在不断变化。如果只有一个 LDTR 寄存器显然不能满足多个 LDT 的要求。

3. 选择子的作用：

   - 选择子是什么？它的值存放在哪里?

     描述符在描述符表中的相对偏移，值存放在段寄存器里。
   - 选择子里面的内容有哪些?

     选择子是一个 2 字节的数，共 16 位，最低 2 位表示 RPL（请求特权等级），第 3 位表示查表是利用 GDT（全局描述符表）还是 LDT（局部描述符表）进行，最高 13 位给出了所需的描述符在描述符表中的地址.
   - 为什么偏移地址大小是 13 位?

     GDTR 是一个 48 位的寄存器，其中 32 位表示段地址，16 位表示段限（最大 64K，每个描述符 8 字节，故最多有 64K/8=8K 个描述符）13 位正好足够寻址 8K 项。

4. 描述符的作用：

   描述一个段是否在内存中，段的起始地址、界限、属性等内容

5. GDTR/LDTR 的作用：

- GDTR 的内容是什么?

  全局描述符表的位置

- LDTR 的内容是什么?

  LDT 的描述符在 GDT 中的相对偏移，根据 GDTR 中的内容即可得到 LDT 的描述符

6. 根目录区大小一定么? 扇区号是多少? 为什么?

   不一定 19 1(引导扇区)+9(FAT1)+9(FAT2) =19

7. 数据区第一个簇号是多少? 为什么?

   第一个簇号为 2，在 1.44M 软盘上，FAT 前三个字节的值必须是固定的，分别是 0xF0、0xFF、0xFF，用于表示这是一个应用在 1.44M 软盘上的 FAT12 文件系统。本来序号为 0 和 1 的 FAT 表项应该对应于簇 0 和簇 1，但是由于这两个表项被设置成了固定值，簇 0 和簇 1 就没有存在的意义了，所以数据区就起始于簇 2。

8. FAT 表的作用?

   记录硬盘中有关文件如何被分散存储在不同扇区的信息（也可以回答为了找到所有的簇 (扇区)）

9. 解释静态链接的过程。

   相似段合并；重定位。

10. 解释动态链接的过程。

    动态链接器自举；装载共享对象；重定位和初始化

11. 静态链接相关 PPT 中为什么使用 ld 链接而不是 gcc。

    使用 ld 进行连接的原因是为了避免 gcc 进行 glibc 的链接

12. linux 下可执行文件的虚拟地址空间默认从哪里开始分配。

    linux 下可执行文件的虚拟空间地址默认从 0x08048000 开始分配

实验相关问题:

1. BPB 指定字段的含义

2. 如何进入子目录并输出 (说明方法调用)

3. 如何获得指定文件的内容, 即如何获得数据区的内容 (比如使用指针等)

4. 如何进行 C 代码和汇编之间的参数传递和返回值传递

5. 汇编代码中对 I/O 的处理方式, 说明指定寄存器所存值的含义

6. 可以要求解释某些看不懂的代码 (我看不懂的话, 你得讲给我听)

思考 上面的程序中，为什么使用 argv 作为基准来定位各个结构的地址，而不是采用 argc？提示：传值和传址。

7.6 动态链接的步骤和实现

有了前面诸多的铺垫，我们终于要开始分析动态链接的实际链接步骤了。动态链接的步骤基本上分为 3 步：先是启动动态链接器本身，然后装载所有需要的共享对象，最后是重定位和初始化。

7.6.1 动态链接器自举

我们知道动态链接器本身也是一个共享对象，但是事实上它有一些特殊性。对于普通共享对象文件来说，它的重定位工作由动态链接器来完成；它也可以依赖于其他共享对象，其中的被依赖的共享对象由动态链接器负责链接和装载。可是对于动态链接器本身来说，它的重定位工作由谁来完成？它是否可以依赖于其他的共享对象？

这是一个“鸡生蛋，蛋生鸡”的问题，为了解决这种无休止的循环，动态链接器这个“鸡”必须有些特殊性。首先是，动态链接器本身不可以依赖于其他任何共享对象；其次是动态链接器本身所需要的全局和静态变量的重定位工作由它本身完成。对于第一个条件我们可以人为地控制，在编写动态链接器时保证不使用任何系统库、运行库；对于第二个条件，动态链接器必须在启动时有一段非常精巧的代码可以完成这项艰巨的工作而同时又不能用到全局和静态变量。这种具有一定限制条件的启动代码往往被称为**自举**（Bootstrap）。

动态链接器入口地址即是自举代码的入口，当操作系统将进程控制权交给动态链接器时，动态链接器的自举代码即开始执行。自举代码首先会找到它自己的 GOT。而 GOT 的第一个入口保存的即是“.dynamic”段的偏移地址，由此找到了动态连接器本身的“.dynamic”段。通过“.dynamic”中的信息，自举代码便可以获得动态链接器本身的重定位表和符号表等，从而得到动态链接器本身的重定位入口，先将它们全部重定位。从这一步开始，动态链接器代码中才可以开始使用自己的全局变量和静态变量。

实际上在动态链接器的自举代码中，除了不可以使用全局变量和静态变量之外，甚至不能调用函数，即动态链接器本身的函数也不能调用。这是为什么呢？其实我们在前面分析地址无关代码时已经提到过，实际上使用 PIC 模式编译的共享对象，对于模块内部的函数调用也是采用跟模块外部函数调用一样的方式，即使用 GOT/PLT 的方式，所以在 GOT/PLT 没有被重定位之前，自举代码不可以使用任何全局变量，也不可以调用函数。下面这段注释来自于 Glibc 2.6.1 源代码中的 elf/rtld.c：

```
/* Now life is sane; we can call functions and access global data.
   Set up to use the operating system facilities, and find out from
   the operating system's program loader where to find the program
   header table in core.  Put the rest of _dl_start into a separate
   function, that way the compiler cannot put accesses to the GOT
   before ELF_DYNAMIC_RELOCATE.  */
```

这段注释写在自举代码的末尾，表示自举代码已经执行结束。“Now life is sane”，可以想象动态链接器的作者在此时大舒一口气，终于完成自举了，可以自由地调用各种函数并且随意访问全局变量了。

7.6.2 装载共享对象

完成基本自举以后，动态链接器将可执行文件和链接器本身的符号表都合并到一个符号表当中，我们可以称它为**全局符号表**（Global Symbol Table）。然后链接器开始寻找可执行文件所依赖的共享对象，我们前面提到过“.dynamic”段中，有一种类型的入口是DT_NEEDED，它所指出的是该可执行文件（或共享对象）所依赖的共享对象。由此，链接器可以列出可执行文件所需要的所有共享对象，并将这些共享对象的名字放入到一个装载集合中。然后链接器开始从集合里取一个所需要的共享对象的名字，找到相应的文件后打开该文件，读取相应的ELF文件头和“.dynamic”段，然后将它相应的代码段和数据段映射到进程空间中。如果这个ELF共享对象还依赖于其他共享对象，那么将所依赖的共享对象的名字放到装载集合中。如此循环直到所有依赖的共享对象都被装载进来为止，当然链接器可以有不同的装载顺序，如果我们把依赖关系看作一个图的话，那么这个装载过程就是一个图的遍历过程，链接器可能会使用深度优先或者广度优先或者其他的顺序来遍历整个图，这取决于链接器，比较常见的算法一般都是广度优先的。

当一个新的共享对象被装载进来的时候，它的符号表会被合并到全局符号表中，所以当所有的共享对象都被装载进来的时候，全局符号表里面将包含进程中所有的动态链接所需要的符号。

符号的优先级

在动态链接器按照各个模块之间的依赖关系，对它们进行装载并且将它们的符号并入到全局符号表时，会不会有这么一种情况发生，那就是有可能两个不同的模块定义了同一个符号？让我们来看看这样一个例子：共有4个共享对象a1.so、a2.so、b1.so和b2.so，它们的源代码文件分别为a1.c、a2.c、b1.c和b2.c：

```
/* a1.c */
#include <stdio.h>

void a()
{
```

```
    printf("a1.c\n");
}

/* a2.c */
#include <stdio.h>

void a()
{
    printf("a2.c\n");
}

/* b1.c */
void a();

void b1()
{
    a();
}

/* b2.c */
void a();

void b2()
{
    a();
}
```

可以看到a1.c和a2.c中都定义了名字为“a”的函数。那么由于b1.c和b2.c都用到了外部函数“a”，但由于源代码中没有指定依赖于哪个共享对象中的函数“a”，所以我们在编译时指定依赖关系。我们假设b1.so依赖于a1.so，b2.so依赖于a2.so，将b1.so与a1.so进行链接，b2.so与a2.so进行链接：

```
$ gcc -fPIC -shared a1.c -o a1.so
$ gcc -fPIC -shared a2.c -o a2.so
$ gcc -fPIC -shared b1.c a1.so -o b1.so
$ gcc -fPIC -shared b2.c a2.so -o b2.so
$ ldd b1.so
        linux-gate.so.1 =>  (0xffffe000)
        a1.so => not found
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e86000)
        /lib/ld-linux.so.2 (0x80000000)
$ldd b2.so
        linux-gate.so.1 =>  (0xffffe000)
        a2.so => not found
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e17000)
        /lib/ld-linux.so.2 (0x80000000)
```

那么当有程序同时使用b1.c中的函数b1和b2.c中的函数b2会怎么样呢？比如有程序main.c：

```
/* main.c */
#include <stdio.h>

void b1();
```

```
void b2();

int main()
{
    b1();
    b2();
    return 0;
}
```

然后我们将 main.c 编译成可执行文件并且运行：

```
$gcc main.c b1.so b2.so -o main -Xlinker -rpath ./
./main
a1.c
a1.c
```

"–XLinker –rpath ./" 表示链接器在当前路径寻找共享对象，否则链接器会报无法找到 a1.so 和 a2.so 错误

很明显，main 依赖于 b1.so 和 b2.so；b1.so 依赖于 a1.so；b2.so 依赖于 a2.so，所以当动态链接器对 main 程序进行动态链接时，b1.so、b2.so、a1.so 和 a2.so 都会被装载到进程的地址空间，并且它们中的符号都会被并入到全局符号表，通过查看进程的地址空间信息可看到：

```
$ cat /proc/14831/maps
08048000-08049000 r-xp 00000000 08:01 1344643    ./main
08049000-0804a000 rwxp 00000000 08:01 1344643    ./main
b7e83000-b7e84000 rwxp b7e83000 00:00 0
b7e84000-b7e85000 r-xp 00000000 08:01 1343481    ./a2.so
b7e85000-b7e86000 rwxp 00000000 08:01 1343481    ./a2.so
b7e86000-b7e87000 r-xp 00000000 08:01 1343328    ./a1.so
b7e87000-b7e88000 rwxp 00000000 08:01 1343328    ./a1.so
b7e88000-b7fcc000 r-xp 00000000 08:01 1488993
/lib/tls/i686/cmov/libc-2.6.1.so
b7fcc000-b7fcd000 r-xp 00143000 08:01 1488993
/lib/tls/i686/cmov/libc-2.6.1.so
b7fcd000-b7fcf000 rwxp 00144000 08:01 1488993
/lib/tls/i686/cmov/libc-2.6.1.so
b7fcf000-b7fd3000 rwxp b7fcf000 00:00 0
b7fde000-b7fdf000 r-xp 00000000 08:01 1344641    ./b2.so
b7fdf000-b7fe0000 rwxp 00000000 08:01 1344641    ./b2.so
b7fe0000-b7fe1000 r-xp 00000000 08:01 1344637    ./b1.so
b7fe1000-b7fe2000 rwxp 00000000 08:01 1344637    ./b1.so
b7fe2000-b7fe4000 rwxp b7fe2000 00:00 0
b7fe4000-b7ffe000 r-xp 00000000 08:01 1455332    /lib/ld-2.6.1.so
b7ffe000-b8000000 rwxp 00019000 08:01 1455332    /lib/ld-2.6.1.so
bfdd2000-bfde7000 rw-p bfdd2000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

这 4 个共享对象的确都被装载进来了，那 a1.so 中的函数 a 和 a2.so 中的函数 a 是不是冲突了呢？为什么 main 的输出结果是两个“a1.c”呢？也就是说 a2.so 中的函数 a 似乎被忽略了。这种一个共享对象里面的全局符号被另一个共享对象的同名全局符号覆盖的现象又被称

为共享对象**全局符号介入**（Global Symbol Interpose）。

关于全局符号介入这个问题，实际上 Linux 下的动态链接器是这样处理的：它定义了一个规则，那就是**当一个符号需要被加入全局符号表时，如果相同的符号名已经存在，则后加入的符号被忽略**。从动态链接器的装载顺序可以看到，它是按照广度优先的顺序进行装载的，首先是 main，然后是 b1.so、b2.so、a1.so，最后是 a2.so。当 a2.so 中的函数 a 要被加入全局符号表时，先前装载 a1.so 时，a1.so 中的函数 a 已经存在于全局符号表，那么 a2.so 中的函数 a 只能被忽略。所以整个进程中，所有对于符合“a”的引用都会被解析到 a1.so 中的函数 a，这也是为什么 main 打印出的结果是两个“a1.c”而不是理想中的“a1.c”和“a2.c”。

由于存在这种重名符号被直接忽略的问题，当程序使用大量共享对象时应该非常小心符号的重名问题，如果两个符号重名又执行不同的功能，那么程序运行时可能会将所有该符号名的引用解析到第一个被加入全局符号表的使用该符号名的符号，从而导致程序莫名其妙的错误。

全局符号介入与地址无关代码

前面介绍地址无关代码时，对于第一类模块内部调用或跳转的处理时，我们简单地将其当作是相对地址调用/跳转。但实际上这个问题比想象中要复杂，结合全局符号介入，关于调用方式的分类的解释会更加清楚。还是拿前面“pic.c”的例子来看，由于可能存在全局符号介入的问题，foo 函数对于 bar 的调用不能够采用第一类模块内部调用的方法，因为一旦 bar 函数由于全局符号介入被其他模块中的同名函数覆盖，那么 foo 如果采用相对地址调用的话，那个相对地址部分就需要重定位，这又与共享对象的地址无关性矛盾。所以对于 bar() 函数的调用，编译器只能采用第三种，即当作模块外部符号处理，bar()函数被覆盖，动态链接器只需要重定位“.got.plt”，不影响共享对象的代码段。

为了提高模块内部函数调用的效率，有一个办法是把 bar()函数变成编译单元私有函数，即使用“static”关键字定义 bar()函数，这种情况下，编译器要确定 bar()函数不被其他模块覆盖，就可以使用第一类的方法，即模块内部调用指令，可以加快函数的调用速度。

7.6.3 重定位和初始化

当上面的步骤完成之后，链接器开始重新遍历可执行文件和每个共享对象的重定位表，将它们的 GOT/PLT 中的每个需要重定位的位置进行修正。因为此时动态链接器已经拥有了进程的全局符号表，所以这个修正过程也显得比较容易，跟我们前面提到的地址重定位的原理基本相同。在前面介绍动态链接下的重定位表时，我们已经碰到过几种重定位类型，每种重定位入口地址的计算方式我们在这里就不再重复介绍了。

重定位完成之后，如果某个共享对象有“.init”段，那么动态链接器会执行“.init”段

中的代码，用以实现共享对象特有的初始化过程，比如最常见的，共享对象中的C++的全局/静态对象的构造就需要通过“.init”来初始化。相应地，共享对象中还可能有“.finit”段，当进程退出时会执行“.finit”段中的代码，可以用来实现类似C++全局对象析构之类的操作。

如果进程的可执行文件也有“.init”段，那么动态链接器不会执行它，因为可执行文件中的“.init”段和“.finit”段由程序初始化部分代码负责执行，我们将在后面的“库”这一部分详细介绍程序初始化部分。

当完成了重定位和初始化之后，所有的准备工作就宣告完成了，所需要的共享对象也都已经装载并且链接完成了，这时候动态链接器就如释重负，将进程的控制权转交给程序的入口并且开始执行。

7.6.4 Linux 动态链接器实现

在前面分析Linux下程序的装载时，已经介绍了一个通过execve()系统调用被装载到进程的地址空间的程序，以及内核如何处理可执行文件。内核在装载完ELF可执行文件以后就返回到用户空间，将控制权交给程序的入口。对于不同链接形式的ELF可执行文件，这个程序的入口是有区别的。对于静态链接的可执行文件来说，程序的入口就是ELF文件头里面的e_entry指定的入口；对于动态链接的可执行文件来说，如果这时候把控制权交给e_entry指定的入口地址，那么肯定是不行的，因为可执行文件所依赖的共享库还没有被装载，也没有进行动态链接。所以对于动态链接的可执行文件，内核会分析它的动态链接器地址（在“.interp”段），将动态链接器映射至进程地址空间，然后把控制权交给动态链接器。

Linux动态链接器是个很有意思的东西，它本身是一个共享对象，它的路径是/lib/ld-linux.so.2，这实际上是个软链接，它指向/lib/ld-x.y.z.so，这个才是真正的动态连接器文件。共享对象其实也是ELF文件，它也有跟可执行文件一样的ELF文件头（包括e_entry、段表等）。动态链接器是个非常特殊的共享对象，它不仅是个共享对象，还是个可执行的程序，可以直接在命令行下面运行：

```
$ /lib/ld-linux.so.2
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]
You have invoked `ld.so', the helper program for shared library executables.
This program usually lives in the file `/lib/ld.so', and special directives
in executable files using ELF shared libraries tell the system's program
loader to load the helper program from this file.  This helper program loads
the shared libraries needed by the program executable, prepares the program
to run, and runs it.  You may invoke this helper program directly from the
command line to load and run an ELF executable file; this is like executing
that file itself, but always uses this helper program from the file you
specified, instead of the helper program file specified in the executable
file you run.  This is mostly of use for maintainers to test new versions
of this helper program; chances are you did not intend to run this program.
```

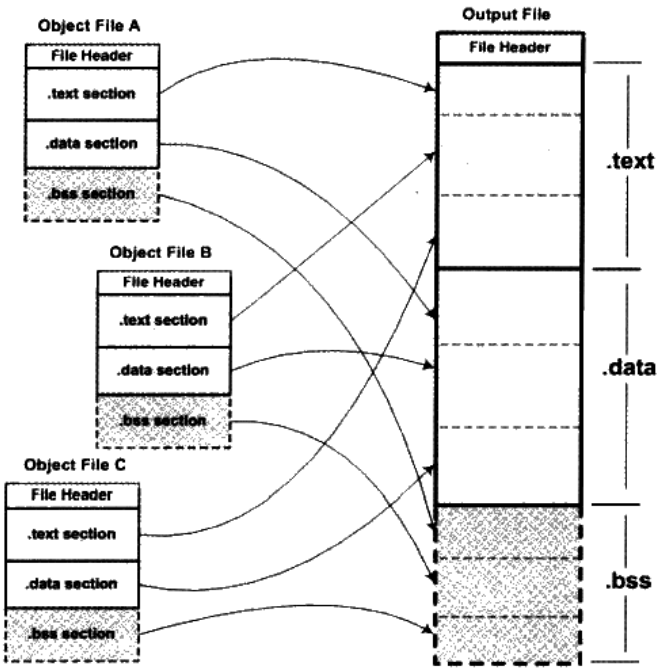

图 4-2 实际的空间分配策略

正如我们前文所提到的，“.bss”段在目标文件和可执行文件中并不占用文件的空间，但是它在装载时占用地址空间。所以链接器在合并各个段的同时，也将“.bss”合并，并且分配虚拟空间。从“.bss”段的空间分配上我们可以思考一个问题，那就是这里的所谓的“空间分配”到底是什么空间？

“链接器为目标文件分配地址和空间”这句话中的“地址和空间”其实有两个含义：第一个是在输出的可执行文件中的空间；第二个是在装载后的虚拟地址中的虚拟地址空间。对于有实际数据的段，比如“.text”和“.data”来说，它们在文件中和虚拟地址中都要分配空间，因为它们在这两者中都存在；而对于“.bss”这样的段来说，分配空间的意义只局限于虚拟地址空间，因为它在文件中并没有内容。**事实上，我们在这里谈到的空间分配只关注于虚拟地址空间的分配**，因为这个关系到链接器后面的关于地址计算的步骤，而可执行文件本身的空间分配与链接过程关系并不是很大。

关于可执行文件和虚拟地址空间之间的关系请参考第 10 章“可执行文件的装载与进程”。

现在的链接器空间分配的策略基本上都采用上述方法中的第二种，使用这种方法的链接器一般都采用一种叫**两步链接**（Two-pass Linking）的方法。也就是说整个链接过程分两步。

第一步 空间与地址分配 扫描所有的输入目标文件，并且获得它们的各个段的长度、属性和位置，并且将输入目标文件中的符号表中所有的符号定义和符号引用收集起来，统一放到一个全局符号表。这一步中，链接器将能够获得所有输入目标文件的段长度，并且将它们合并，计算出输出文件中各个段合并后的长度与位置，并建立映射关系。

第二步 符号解析与重定位 使用上面第一步中收集到的所有信息，读取输入文件中段的数据、重定位信息，并且进行符号解析与重定位、调整代码中的地址等。事实上第二步是链接过程的核心，特别是重定位过程。

我们使用 ld 链接器将“a.o”和“b.o”链接起来：

```
$ld a.o b.o -e main -o ab
```

- -e main 表示将 main 函数作为程序入口，ld 链接器默认的程序入口为_start。
- -o ab 表示链接输出文件名为 ab，默认为 a.out。

让我们使用 objdump 来查看链接前后地址的分配情况，代码如清单 4-1 所示。

清单 4-1 链接前后各个段的属性

```
$ objdump -h a.o
...
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000034  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000068  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000068  2**2
                  ALLOC
...
$ objdump -h b.o
...
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000003e  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  00000000  00000000  00000074  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000078  2**2
                  ALLOC
...
$objdump -h ab
...
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000072  08048094  08048094  00000094  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  08049108  08049108  00000108  2**2
                  CONTENTS, ALLOC, LOAD, DATA
...
```

VMA 表示 Virtual Memory Address，即虚拟地址，LMA 表示 Load Memory Address，即加载地址，正常情况下这两个值应该是一样的，但是在有些嵌入式系统中，特别是在那些程序放在 ROM 的系统中时，LMA 和 VMA 是不相同的。这里我们只要关注 VMA 即可。

链接前后的程序中所使用的地址已经是程序在进程中的虚拟地址，即我们关心上面各个段中的 VMA（Virtual Memory Address）和 Size，而忽略**文件偏移**（File off）。我们可以看到，在链接之前，目标文件中的所有段的 VMA 都是 0，因为虚拟空间还没有被分配，所以它们默认都为 0。等到链接之后，可执行文件“ab”中的各个段都被分配到了相应的虚拟地址。这里的输出程序“ab”中，“.text”段被分配到了地址 0x08048094，大小为 0x72 字节；“.data”段从地址 0x08049108 开始，大小为 4 字节。整个链接过程前后，目标文件各段的分配、程序虚拟地址如图 4-3 所示。

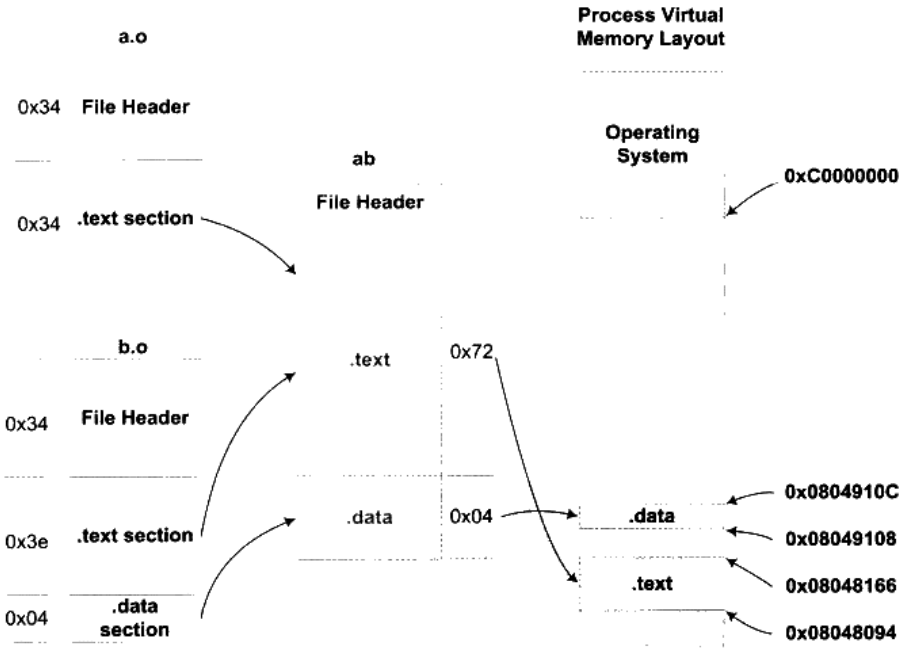

图 4-3 目标文件、可执行文件与进程空间

我们在图 4-3 中忽略了像.comment 这种无关紧要的段，只关心代码段和数据段。由于在本例中没有“.bss”段，所以我们也将其简化了。从图 4-3 中可以看到，“a.o”和“b.o”的代码段被先后叠加起来，合并成“ab”的一个.text 段，加起来的长度为 0x72。所以“ab”的代码段里面肯定包含了 main 函数和 swap 函数的指令代码。

那么，为什么链接器要将可执行文件“ab”的“.text”分配到0x08048094、将“.data”分配0x08049108？而不是从虚拟空间的0地址开始分配呢？这涉及操作系统的进程虚拟地址空间的分配规则，在Linux下，ELF可执行文件默认从地址0x08048000开始分配。关于进程的虚拟地址分配等相关内容我们将在第6章“可执行文件的装载与进程”这一章进行详细的分析。

4.1.3 符号地址的确定

我们还是以“a.o”和“b.o”作为例子，来分析这两个步骤中链接器的工作过程。在第一步的扫描和空间分配阶段，链接器按照前面介绍的空间分配方法进行分配，这时候输入文件中的各个段在链接后的虚拟地址就已经确定了，比如“.text”段起始地址为0x08048094，“.data”段的起始地址为0x08049108。

当前面一步完成之后，链接器开始计算各个符号的虚拟地址。因为各个符号在段内的相对位置是固定的，所以这时候其实“main”、“shared”和“swap”的地址也已经是确定的了，只不过链接器须要给每个符号加上一个偏移量，使它们能够调整到正确的虚拟地址。比如我们假设“a.o”中的“main”函数相对于“a.o”的“.text”段的偏移是X，但是经过链接合并以后，“a.o”的“.text”段位于虚拟地址0x08048094，那么“main”的地址应该是0x08048094+X。从前面“objdump”的输出看到，“main”位于“a.o”的“.text”段的最开始，也就是偏移为0，所以“main”这个符号在最终的输出文件中的地址应该是0x08048094+0，即0x08048094。我们也可以通过完全一样的计算方法得知所有符号的地址，在这个例子里面，只有三个全局符号，所以链接器在更新全局符号表的符号地址以后，各个符号的最终地址如表4-1所示。

表4-1

符号	类型	虚拟地址
main	函数	0x08048094
swap	函数	0x080480c8
shared	变量	0x08048108

4.2 符号解析与重定位

4.2.1 重定位

在完成空间和地址的分配步骤以后，链接器就进入了符号解析与重定位的步骤，这也是静态链接的核心内容。在分析符号解析和重定位之前，首先让我们来看看“a.o”里面是怎

么使用这两个外部符号的，也就是说我们在“a.c”的源程序里面使用了“shared”变量和“swap”函数，那么编译器在将“a.c”编译成指令时，它如何访问“shared”变量？如何调用“swap”函数？

使用 objdump 的“-d”参数可以看到“a.o”的代码段反汇编结果：

```
$objdump -d a.o

a.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  0xfffffffc(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   83 ec 24                sub    $0x24,%esp
  11:   c7 45 f8 64 00 00 00    movl   $0x64,0xfffffff8(%ebp)
  18:   c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
  1f:   00
  20:   8d 45 f8                lea    0xfffffff8(%ebp),%eax
  23:   89 04 24                mov    %eax,(%esp)
  26:   e8 fc ff ff ff          call   27 <main+0x27>
  2b:   83 c4 24                add    $0x24,%esp
  2e:   59                      pop    %ecx
  2f:   5d                      pop    %ebp
  30:   8d 61 fc                lea    0xfffffffc(%ecx),%esp
  33:   c3                      ret
```

我们知道在程序的代码里面使用的都是虚拟地址，在这里也可以看到“main”的起始地址为 0x00000000，这是因为在未进行前面提到过的空间分配之前，目标文件代码段中的起始地址以 0x00000000 开始，等到空间分配完成以后，各个函数才会确定自己在虚拟地址空间中的位置。

我们可以很清楚地看到“a.o”的反汇编结果中，“a.o”共定义了一个函数 main。这个函数占用 0x33 个字节，共 17 条指令；最左边那列是每条指令的偏移量，每一行代表一条指令（有些指令的长度很长，如第偏移为 0x18 的 mov 指令，它的二进制显示占据了两行）。我们已经用粗体标出了两个引用“shared”和“swap”的位置，对于“shared”的引用是一条“mov”指令，这条指令总共 8 个字节，它的作用是将“shared”的地址赋值到 ESP 寄存器+4 的偏移地址中去，前面 4 个字节是指令码，后面 4 个字节是“shared”的地址，我们只关心后面的 4 个字节部分，如图 4-4 所示。

当源代码“a.c”在被编译成目标文件时，编译器并不知道“shared”和“swap”的地址，因为它们定义在其他目标文件中。所以编译器就暂时把地址 0 看作是“shared”的地址，我

们可以看到这条“mov”指令中，关于“shared”的地址部分为“0x00000000”。

图 4-4 绝对地址指令

另外一个是偏移为 0x26 的指令的一条调用指令，它其实就表示对 swap 函数的调用，如图 4-5 所示。

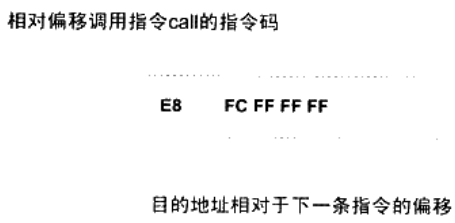

图 4-5 相对地址指令

这条指令共 5 个字节，前面的 0xE8 是操作码（Operation Code），从 Intel 的 IA-32 体系软件开发者手册（IA-32 Intel Architecture Software Developer's Manual，参考文献里有详细介绍）可以查阅到，这条指令是一条**近址相对位移调用指令**（Call near, relative, displacement relative to next instruction），后面 4 个字节就是被调用函数的相对于**调用指令的下一条指令**的偏移量。在没有重定位之前，相对偏移被置为 0xFFFFFFFC（小端），它是常量“–4”的补码形式。

让我们来仔细看这条指令的含义。紧跟在这条 call 指令后面的那条指令为 add 指令，add 指令的地址为 0x2b，而相对于 add 指令偏移为“–4”的地址即 0x2b – 4 = 0x27。所以这条 call 指令的实际调用地址为 0x27。我们可以看到 0x27 存放着并不是 swap 函数的地址，跟前面“shared”一样，“0xFFFFFFFC”只是一个临时的假地址，因为在编译的时候，编译器并不知道“swap”的真正地址。

编译器把这两条指令的地址部分暂时用地址“0x00000000”和“0xFFFFFFFC”代替着，把真正的地址计算工作留给了链接器。我们通过前面的空间与地址分配可以得知，链接器在完成地址和空间分配之后就已经可以确定所有符号的虚拟地址了，那么链接器就可以根据符

号的地址对每个需要重定位的指令进行地位修正。我们用 objdump 来反汇编输出程序“ab”的代码段，可以看到 main 函数的两个重定位入口都已经被修正到正确的位置：

```
$objdump -d ab
ab:     file format elf32-i386

Disassembly of section .text:

08048094 <main>:
 8048094:    8d 4c 24 04             lea     0x4(%esp),%ecx
 8048098:    83 e4 f0                and    $0xfffffff0,%esp
 804809b:    ff 71 fc                pushl  0xfffffffc(%ecx)
 804809e:    55                      push   %ebp
 804809f:    89 e5                   mov    %esp,%ebp
 80480a1:    51                      push   %ecx
 80480a2:    83 ec 24                sub    $0x24,%esp
 80480a5:    c7 45 f8 64 00 00 00    movl   $0x64,0xfffffff8(%ebp)
 80480ac:    c7 44 24 04 08 91 04    movl   $0x8049108,0x4(%esp)
 80480b3:    08
 80480b4:    8d 45 f8                lea     0xfffffff8(%ebp),%eax
 80480b7:    89 04 24                mov    %eax,(%esp)
 80480ba:    e8 09 00 00 00          call   80480c8 <swap>
 80480bf:    83 c4 24                add    $0x24,%esp
 80480c2:    59                      pop    %ecx
 80480c3:    5d                      pop    %ebp
 80480c4:    8d 61 fc                lea     0xfffffffc(%ecx),%esp
 80480c7:    c3                      ret

080480c8 <swap>:
 80480c8:    55                      push   %ebp
...
```

经过修正以后，“shared”和“swap”的地址分别为 0x08049108 和 0x00000009（小端字节序）。关于“shared”很好理解，因为“shared”变量的地址的确是 0x08049108。对于“swap”来说稍显晦涩。我们前面介绍过，这个“call”指令是一条**近址相对位移调用指令**，它后面跟的是**调用指令的下一条指令**的偏移量，“call”指令的下一条指令是“add”，它的地址是 0x080480bf，所以“相对于 add 指令偏移量为 0x00000009”的地址为 0x080480bf + 9 = 0x080480c8，即刚好是“swap ”函数的地址。有兴趣的读者可以阅读后面的“指令修正方式”一节，那里我们将更加详细介绍指令修正时的地址计算方式。

4.2.2 重定位表

那么链接器是怎么知道哪些指令是要被调整的呢？这些指令的哪些部分要被调整？怎么调整？比如上面例子中“mov”指令和“call”指令的调整方式就有所不同。事实上在 ELF 文件中，有一个叫**重定位表**（Relocation Table）的结构专门用来保存这些与重定位相关的信息，我们在前面介绍 ELF 文件结构时已经提到过了重定位表，它在 ELF 文件中往往是一个或多个段。

对于可重定位的 ELF 文件来说，它必须包含有重定位表，用来描述如何修改相应的段里的内容。对于每个要被重定位的 ELF 段都有一个对应的重定位表，而一个重定位表往往就是 ELF 文件中的一个段，所以其实**重定位表**也可以叫**重定位段**，我们在这里统一称作**重定位表**。比如代码段“.text”如有要被重定位的地方，那么会有一个相对应叫“.rel.text”的段保存了代码段的重定位表；如果代码段“.data”有要被重定位的地方，就会有一个相对应叫“.rel.data”的段保存了数据段的重定位表。我们可以使用 objdump 来查看目标文件的重定位表：

```
$ objdump -r a.o

a.o:     file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000001c R_386_32          shared
00000027 R_386_PC32        swap
```

这个命令可以用来查看“a.o”里面要重定位的地方，即“a.o”所有引用到外部符号的地址。每个要被重定位的地方叫一个**重定位入口**（Relocation Entry），我们可以看到“a.o”里面有两个重定位入口。重定位入口的**偏移**（Offset）表示该入口在要被重定位的段中的位置，“RELOCATION RECORDS FOR [.text]”表示这个重定位表是**代码段的重定位表**，所以偏移表示代码段中须要被调整的位置。对照前面的反汇编结果可以知道，这里的 0x1c 和 0x27 分别就是代码段中“mov”指令和“call”指令的地址部分。

对于 32 位的 Intel x86 系列处理器来说，重定位表的结构也很简单，它是一个 Elf32_Rel 结构的数组，每个数组元素对应一个重定位入口。Elf32_Rel 的定义如下：

```
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