2.1 Linux 0.11 支持的目标文件

为了生成内核代码文件，Linux 0.11 使用了两种编译器。第一种是汇编编译器 as86 和相应的链接程序 ld86。它们专门用于编译和链接运行在实地址模式下的 16 位内核引导扇区程序 bootsect.s 和设置程序 setup.s。第二种是 GNU 的汇编器 gas 和 C 语言编译器 gcc 以及相应的链接程序 gld。编译器用于为源程序文件产生含有生成的二进制代码和数据的目标文件。链接程序用于对相关的所有目标文件进行组合处理，形成一个可被内核加载执行的目标文件，即可执行文件。

下面我们首先简单说明编译器产生的目标文件结构，然后描述链接器如何把需要链接在一起的目标文件模块组合在一起，以生成二进制可执行映像文件或一个大的模块文件。最后说明 Linux 0.11 内核二进制代码文件 Image 的生成原理和过程。有关目标文件和链接程序的基本工作原理可参见 John R. Levine 著的《Linkers & Loaders》一书，这里仅给出了能够理解编译链接所生成的 Linux 0.11 内核代码文件的信息以及 Linux 0.11 内核所支持的 a.out 目标文件格式。as86 和 ld86 生成的是 MINIX 专门的目标文件格式，因为其结构与 a.out 目标文件格式类似，因此这里就不再说明。

为便于描述，这里把编译器生成的目标文件称为目标模块文件（简称模块文件），而把链接程序输出产生的可执行目标文件称为可执行文件。并且把它们都统称为目标文件。

2.1.1 目标文件格式

C 语言编译器 gcc 和汇编器 gas 编译生成的 a.out 格式的目标模块文件或链接生成的可执行文件含有 7 个部分：

a) **执行头部分**(exec header)。执行文件头部分。该部分中含有一些参数（exec 结构），是有关目标文件的整体结构信息。例如代码和数据部分的长度、未初始化数据区的长度、对应源程序文件名以及目标文件创建时间等。内核使用这些参数把执行文件加载到内存中并执行，而链接程序（ld）使用这些参数将一些模块文件组合成一个可执行文件。这是目标文件唯一必要的组成部分。

b) **代码段部分**(text segment)。由编译器或汇编器生成的二进制指令代码和数据信息，含有程序执行时被加载到内存中的指令代码和相关数据。可以以只读形式被加载。

c) **数据段部分**(data segment)。由编译器或汇编器生成的二进制指令代码和数据信息，这部分含有已经初始化过的数据，总是被加载到可读写的内存中。

d) **代码重定位部分**(text relocations)。这部分含有供链接程序使用的记录数据。在组合目标模块文件时用于定位代码段中的指针或地址。当链接程序需要改变目标代码的地址时就需要修正和维护这些地方。

e) **数据重定位部分**(data relocations)。类似于代码重定位部分的作用，但是用于数据段中指针的重定位。

f) **符号表部分**(simbol table)。这部分同样含有供链接程序使用的记录数据。这些记录数据保存着模块文件中定义的全局符号以及需要从其他模块文件中输入的符号，或者是由链接器定义的符号，用于在模块文件之间对命名的变量和函数（符号）进行交叉引用。

g) **字符串表部分**(string table)。该部分含有与符号名相对应的字符串。用于调试程序调试目标代码，与链接过程无关。这些信息可包含源程序代码和行号、局部符号以及数据结构描述信息等。

在 Linux 0.11 系统中，GNU gcc 或 gas 编译输出的目标模块文件和链接程序所生成的可执行文件都使用了 UNIX 传统的 a.out 格式。这是一种被称为汇编与链接输出（Assembly & linker editor output）的目标文件格式。对于具有内存分页机制的系统来说，这是一种简单有效的目标文件格式。a.out 格式文件由一个文件头和随后的代码部分（Text section，也称为文本段）、已初始化数据部分（Data section，也称为数据段）、重定位信息部分、符号表以及符号名字符串构成，见图 1 所示。其中代码部分和数据部分通常也被分别称为文本段（代码段）和数据段。

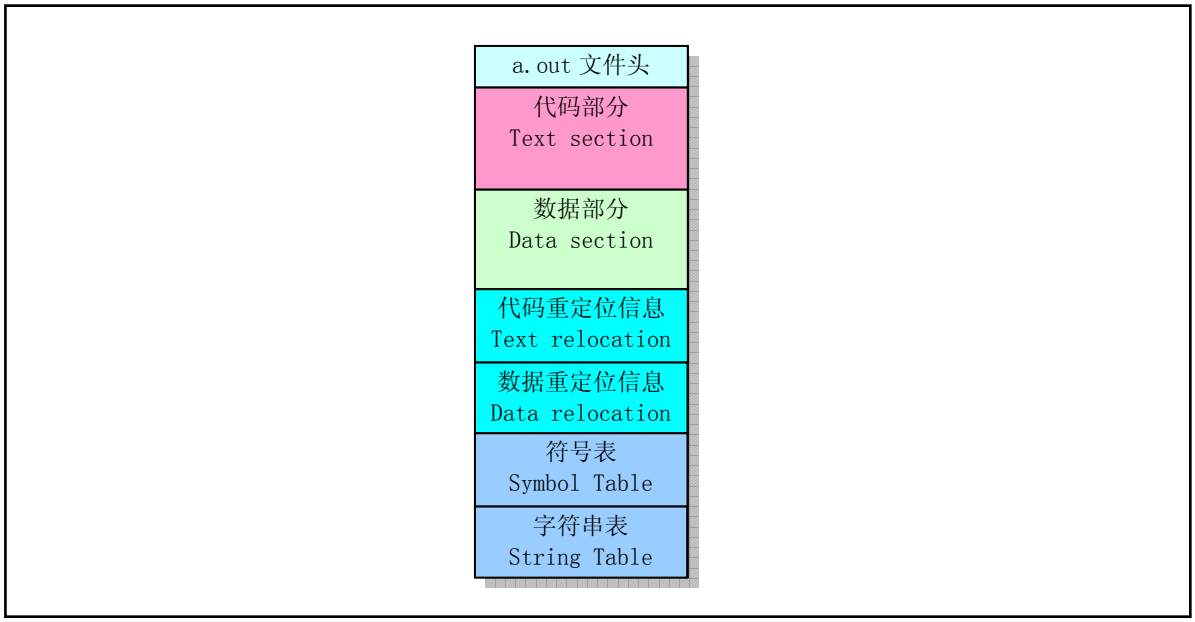

图 1 a.out 格式的目标文件

一个指定的目标文件并非一定会包含所有以上信息。由于 Linux 0.11 系统使用了 Intel CPU 的内存管理功能，因此它会为每个执行程序单独分配一个 64MB 的地址空间（逻辑地址空间）使用。在这种情况下因为链接器已经把执行文件处理成从一个固定地址开始运行，所以相关的可执行文件中就不再需要重定位信息。

目标文件的文件头中含有一个长度为 32 字节的 exec 数据结构，通常称为文件头结构或执行头结构。其定义如下所示。有关 a.out 结构的详细信息请参见 include/a.out.h 文件后的介绍。

```
struct exec {
  unsigned long a_magic         // 执行文件魔数。使用 N_MAGIC 等宏访问。
  unsigned a_text               // 代码长度，字节数。
  unsigned a_data               // 数据长度，字节数。
  unsigned a_bss                // 文件中的未初始化数据区长度，字节数。
  unsigned a_syms               // 文件中的符号表长度，字节数。
  unsigned a_entry              // 执行开始地址。
  unsigned a_trsize             // 代码重定位信息长度，字节数。
  unsigned a_drsize             // 数据重定位信息长度，字节数。
}
```

根据 a.out 文件中头结构魔数字段的值，我们又可把 a.out 格式的文件分成几种类型。Linux 0.11 系统使用了其中两种类型：模块目标文件使用了 OMAGIC 类型的 a.out 格式，其魔数是 0x107（八进制 0407）。而执行文件则使用了 ZMAGIC 类型的 a.out 格式，魔数是 0x10b（八进制 0413）。这两种格式的主要区别在于它们对各个部分的存储分配方式上。虽然该结构的总长度只有 32 字节，但是对于一个 ZMAGIC 类型的执行文件来说，其文件开始部分却需要专门留出 1024 字节的空间给头结构使用。除被头结构占用的 32 个字节以外，其余部分均为 0。从 1024 字节之后才开始放置程序的文本段和数据段等信息。而对于一个 OMAGIC 类型的.o 模块文件来说，文件开始部分的 32 字节头结构后面紧接着就是代码部分和数据部分。

执行头结构中 a_text 和 a_data 字段分别指出了后面只读的代码段和可读写数据段的字节长度。a_bss 字段指明内核在加载目标文件时数据段后面未初始化数据区域（bss 段）的长度。由于 Linux 在分配内存时会自动对内存清零，因此 bss 段不需要被包括在模块文件或执行文件中。为了形象地表示目标文件逻辑地具有一个 bss 段，在后面图示中将使用虚线框来表示目标文件中的 bss 段。

a_entry 字段指定了程序开始执行的地址，而 a_syms、a_trsize 和 a_drsize 字段则分别说明了数据段后符号表、代码和数据段重定位信息的大小。对于可执行文件来说并不需要符号表和重定位信息，因此除非链接程序为了调试目的而包括了符号信息，执行文件中的这几个字段的值通常为 0。

Linux 0.11 系统的模块文件和执行文件都是 a.out 格式的目标文件，但是只有编译器生成的模块文件中包含用于链接程序的重定位信息。代码段和数据段的重定位信息均有重定位记录（项）构成，每个记录的长度为 8 字节，其结构如下所示。

```
struct relocation_info
{
  int r_address;                    // 段内需要重定位的地址。
  unsigned int r_symbolnum:24;      // 含义与 r_extern 有关。指定符号表中一个符号或者一个段。
  unsigned int r_pcrel:1;           // 1 比特。PC 相关标志。
  unsigned int r_length:2;          // 2 比特。指定要被重定位字段长度（2 的次方）。
  unsigned int r_extern:1;          // 外部标志位。1 - 以符号的值重定位。0 - 以段的地址重定位。
  unsigned int r_pad:4;             // 没有使用的 4 个比特位，但最好将它们复位掉。
};
```

重定位项的功能有两个。一是当代码段被重定位到一个不同的基地址处时，重定位项则用于指出需要修改的地方。二是在模块文件中存在对未定义符号的引用时，当此未定义符号最终被定义时链接程序就可以使用相应重定位项对符号的值进行修正。由上面重定位记录项的定义可以看出，每个记录项含有模块文件代码部分（代码段）和数据部分（数据段）中需要重定位处长度为 4 字节的地址以及规定如何具体进行重定位操作的信息。地址字段 r_address 是指可重定位项从代码段或数据段开始算起的偏移值。2 比特的长度字段 r_length 指出被重定位项的长度，0 到 3 分别表示被重定位项的宽度是 1 字节、2 字节、4 字节或 8 字节。标志位 r_pcrel 指出被重定位项是一个“PC 相关的”的项，即它作为一个相对地址被用于指令当中。外部标志位 r_extern 控制着 r_symbolnum 的含义，指明重定位项参考的是段还是一个符号。如果该标志值是 0，那么该重定位项是一个普通的重定位项，此时 r_symbolnum 字段指定是在哪个段中寻址定位。如果该标志是 1，那么该重定位项是对一个外部符号的引用，此时 r_symbolnum 指定目标文件中符号表中的一个符号，需要使用符号的值进行重定位。

目标文件的最后一部分是符号表和相关的字符串表。符号表记录项的结构如下所示。

```
struct nlist {
  union {
    char         *n_name;           // 字符串指针，
    struct nlist *n_next;           // 或者是指向另一个符号项结构的指针，
    long          n_strx;           // 或者是符号名称在字符串表中的字节偏移值。
  } n_un;
  unsigned char n_type;             // 该字节分成 3 个字段，参见 a.out.h 文件 146-154 行。
  char          n_other;            // 通常不用。
  short         n_desc;             //
  unsigned long n_value;            // 符号的值。
};
```

由于 GNU gcc 编译器允许任意长度的标识符，因此标识符字符串都位于符号表后的字符串表中。每个符号表记录项长度为 12 字节，其中第一个字段给出了符号名字符串（以 null 结尾）在字符串表中的偏移位置。类型字段 n_type 指明了符号的类型。该字段的最后一个比特位用于指明符号是否是外部的（全局的）。如果该位为 1 的话，那么说明该符号是一个全局符号。链接程序并不需要非全局符号信息，但可供调试程序使用。n_type 字段的其余比特位用来指明符号类型。a.out.h 头文件中定义了这些类型值常量符号。符号的主要的类型包括：

- text、data 或 bbs 指明是本模块文件中定义的符号。此时符号的值是模块中该符号的可重定位地址。
- abs 指明符号是一个绝对的（固定的）不可重定位的符号。符号的值就是该固定值。
- undef 指明是一个本模块文件中未定义的符号。此时符号的值通常是 0。

但作为一种特殊情况，编译器能够使用一个未定义的符号来要求链接程序为指定的符号名保留一块存储空间。如果一个未定义的外部（全局）符号具有非零值，那么对链接程序而言该值就是程序希望指定符号寻址的存储空间的大小值。在链接操作期间，如果该符号确实没有定义，那么链接程序就会在 bss 段中为该符号名建立一块存储空间，空间的大小是所有被链接模块中该符号值最大的一个。这个就是 bss 段中所谓的公共块（Common block）定义，主要用于支持未初始化的外部（全局）数据。例如程序中定义的未初始化的数组。如果该符号在任意一个模块中已经被定义了，那么链接程序就会使用该定义而忽略该值。

在 Linux 0.11 系统中，我们可以使用 objdump 命令来查看模块文件或执行文件中文件头结构的具体值。例如，下面列出了 hello.o 目标文件及其执行文件中文件头的具体值。

```
[/usr/root]# gcc -c -o hello.o hello.c
[/usr/root]# gcc -o hello hello.o
[/usr/root]#
[/usr/root]# objdump
Usage: objdump [-hnrt] [+header] [+nstuff] [+relocation] [+symbols] objfile...
[/usr/root]#
[/usr/root]# objdump -h hello.o
hello.o:
magic: 0x107 (407)machine type: 0 flags: 0x0 text 0x28 data 0x0 bss 0x0
nsyms 3 entry 0x0 trsize 0x10 drsize 0x0
[/usr/root]#
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0 flags: 0x0 text 0x3000 data 0x1000 bss 0x0
nsyms 141 entry 0x0 trsize 0x0 drsize 0x0
[/usr/root]#
```

可以看出，hello.o 模块文件的魔数是 0407（OMAGIC），除了文件头结构以外，还包括一个长度为 0x28 字节的代码段和一个具有 3 个符号项的符号表以及长度为 0x10 字节的代码段重定位信息。其余各段的长度均为 0。对应的执行文件 hello 的魔数是 0413（ZMAGIC），代码段和数据段的长度分别为 0x3000 和 0x1000 字节，并带有包含 141 个项的符号表。我们可以使用命令 strip 删除执行文件中的符号表信息。例如下面我们删除了 hello 执行文件中的符号信息。可以看出 hello 执行文件的符号表长度变成了 0，并且 hello 文件的长度也从原来的 20591 字节减小到 17412 字节。

```
[/usr/root]# ll hello
-rwx--x--x   1 root     4096        20591 Nov 14 18:30 hello
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0flags: 0x0text 0x3000 data 0x1000 bss 0x0
nsyms 141 entry 0x0 trsize 0x0 drsize 0x0

[/usr/root]# strip hello
[/usr/root]# ll hello
-rwx--x--x   1 root     4096        17412 Nov 14 18:33 hello
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0flags: 0x0text 0x3000 data 0x1000 bss 0x0
```

```
nsyms 0 entry 0x0 trsize 0x0 drsize 0x0
[/usr/root]#
```

磁盘上 a.out 执行文件的各部分在进程逻辑地址空间中的对应关系见图 2 所示。Linux 0.11 系统中进程的逻辑空间大小是 64MB。对于 ZMAGIC 类型的 a.out 执行文件，它的代码部分的长度是内存页面的整数倍。由于 Linux 0.11 内核使用需求页（Demand-paging）技术，即在一页代码实际要使用的时候才被加载到物理内存页面中，而在进行加载操作的 fs/execve()函数中仅仅为其设置了分页机制的页目录项和页表项，因此需求页技术可以加快程序的加载的速度。

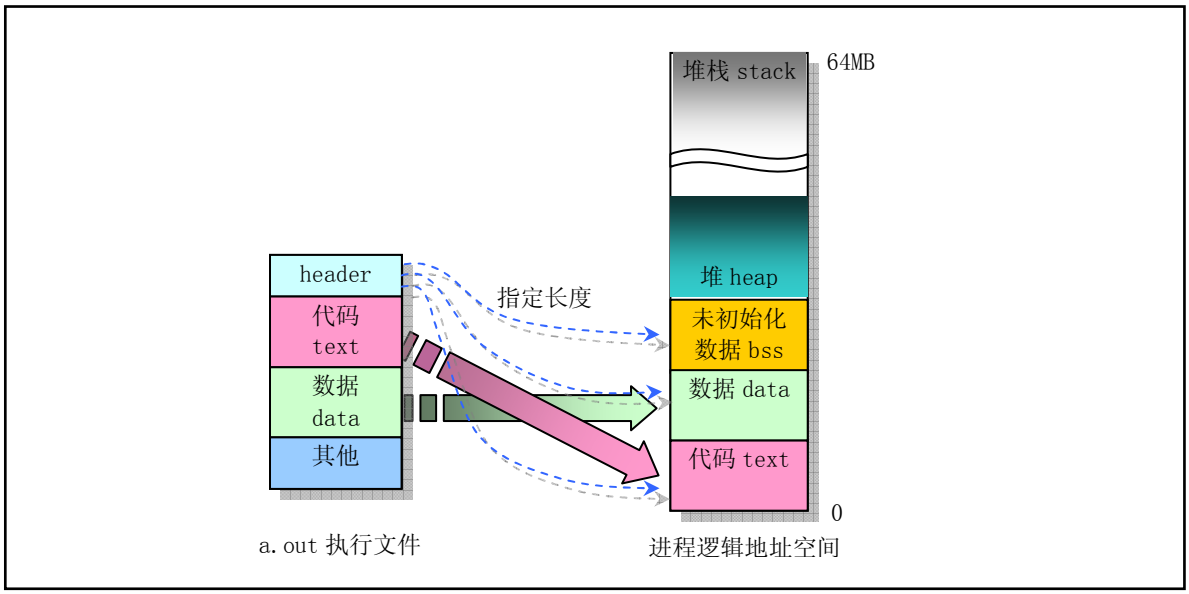

图 2 a.out 执行文件映射到进程逻辑地址空间

图中 bss 是进程的未初始化数据部分，用于存放静态的未初始化数据。在开始执行程序时 bss 的第 1 页内存会被设置为全 0。图中 heap 是堆空间部分，用于分配进程在执行过程中动态申请的内存空间。

2.1.2 链接程序输出

链接程序对输入的一个或多个模块文件以及相关的库函数模块进行处理，最终生成相应的二进制执行文件或者是一个所有模块组合成的大模块文件。在这个过程中，链接程序的首要任务是给执行文件（或者输出的模块文件）进行存储空间分配操作。一旦存储位置确定，链接程序就可以继续执行符号邦定操作和代码修正操作。因为模块文件中定义的大多数符号与文件中的存储位置有关，所以在符号对应的位置没有确定下来之前符号是没有办法解析的。

每个模块文件中包括几种类型的段，链接程序的第二个任务就是把所有模块中相同类型的段组合连接在一起，在输出文件中为指定段类型形成单一一个段。例如，链接程序需要把所有输入模块文件中的代码段合并成一个段放在输出的执行文件中。

对于 a.out 格式的模块文件来说，由于段类型是预先知道的，因此链接程序对 a.out 格式的模块文件进行存储分配比较容易。例如，对于具有两个输入模块文件和需要连接一个库函数模块的情况，其存储分配情况见图 3 所示。每个模块文件都有一个代码段（text）、数据段（data）和一个 bss 段，也许还会有一些看似外部（全局）符号的公共块。链接程序会收集每个模块文件包括任何库函数模块中的代码段、数据段和 bss 段的大小。在读入并处理了所有模块之后，任何具有非零值的未解析的外部符号都将作为公共块来看待，并且把它们分配存储在 bss 段的末尾处。

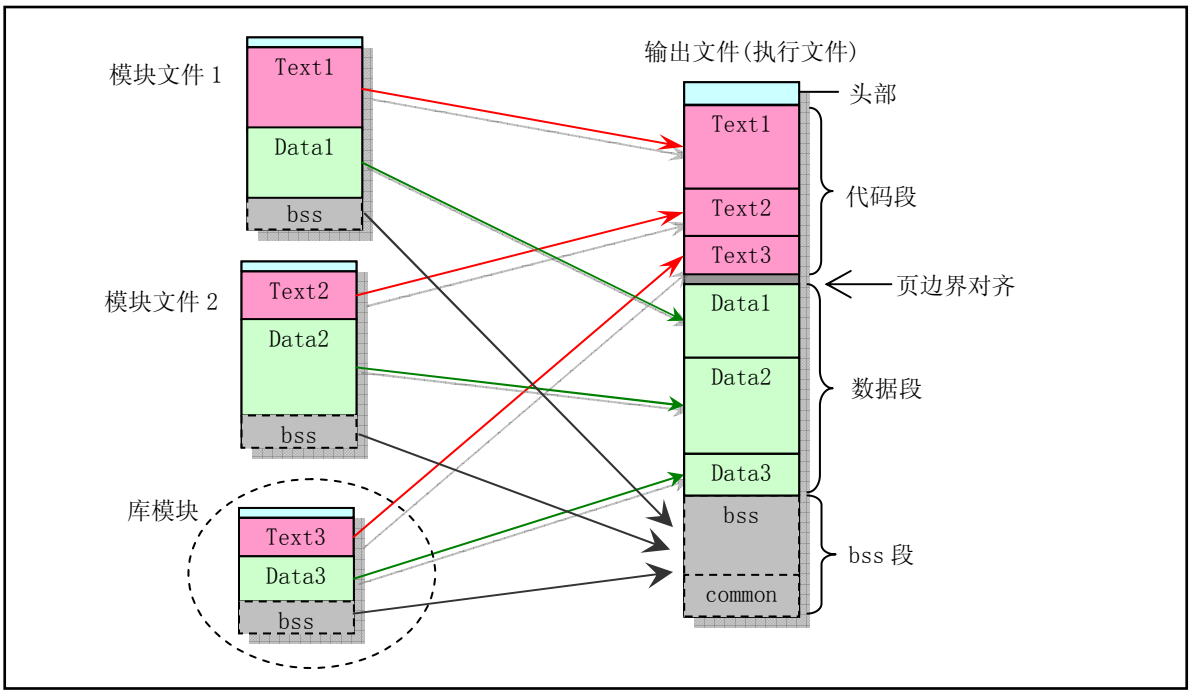

图 3 目标文件的链接操作

此后链接程序就可以为所有段分配地址。对于 Linux 0.11 系统中使用的 ZMAGIC 类型的 a.out 格式，输出文件中的代码段被设置成从固定地址 0 开始。数据段则从代码段后下一个页面边界开始。bss 段则紧随数据段开始放置。在每个段内，链接程序会把输入模块文件中的同类型段顺序存放，并按字进行边界对齐。

当 Linux 0.11 内核加载一个可执行文件时，它会根据文件头部结构中的信息首先判断文件是否是一个合适的可执行文件，即其魔数类型是否为 ZMAGIC，然后系统在用户态堆栈顶部为程序设置环境参数和命令行上输入的参数信息块并为其构建一个任务数据结构。接着在设置了一些相关寄存器值后利用堆栈返回技术去执行程序。

对于 Linux 0.11 内核的编译过程，它是根据内核的配置文件 Makefile 使用 make 命令指挥编译器和链接程序操作而完成的。在建立过程中 make 还利用内核源代码 tools/目录下的 build.c 程序编译生成了一个用于组合所有模块的临时工具程序 build。由于内核是由引导启动程序利用 ROM BIOS 中断调用加载到内存中，因此编译产生的内核各模块中的执行头结构部分需要去掉。工具程序 build 主要功能就是分别去掉 bootsect、setup 和 system 文件中的执行头结构，然后把它们的数据和代码顺序组合在一起产生一个名为 Image 的内核映象文件。

--- end ---