# 4.处理器体系结构

前3节所描述的计算机系统仅限于机器语言程序级，而这些指令是由处理器来执行的。**一个处理器支持的指令和指令的字节级编码称为它的指令集体系结构（ISA）**。不同的处理器家族，例如Intel IA32和x86-64、IBM/Freescale Power和ARM处理器家族，都有不同的ISA。

需要知道的是，现代的处理器工作方式可能跟ISA所隐含的计算模型**大相径庭**。ISA模型看上去应该是**顺序**指令执行，而**同时**处理多条指令的不同部分可以使处理器获得更高的性能。本节将首先定义一个简单的指令集，称为"Y86-64"指令集，与x86-64相比，其数据类型、指令和寻址方式等都要少一点，但是依然足够完整。

## 4.1Y86-64指令集体系结构

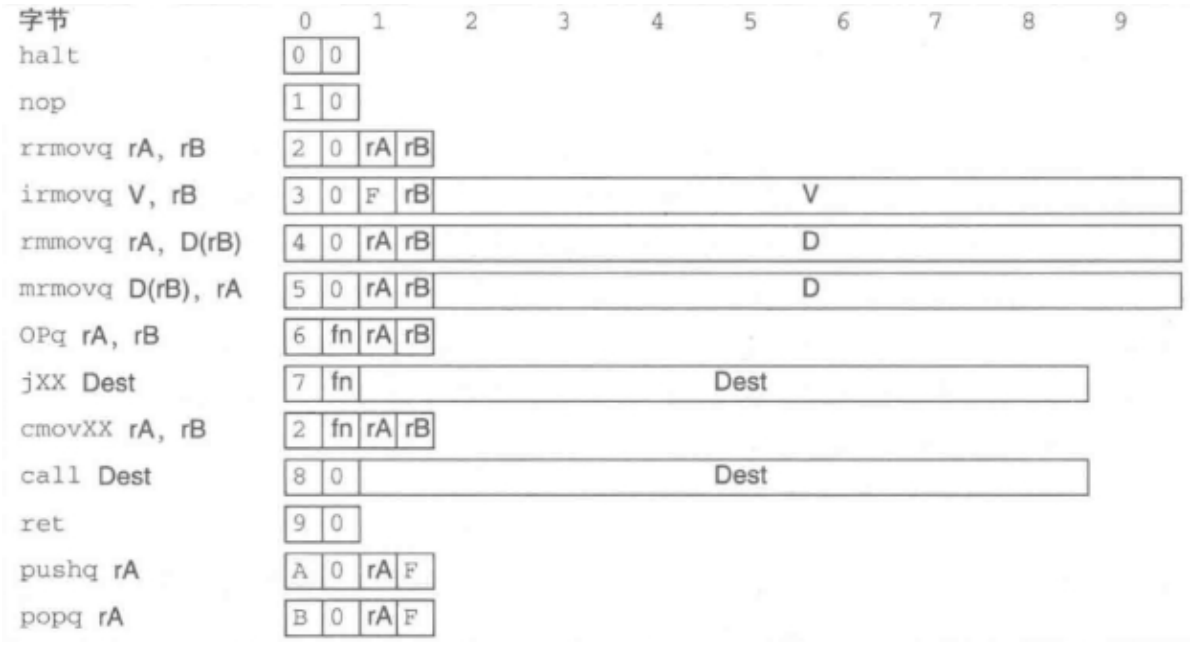
定义一个指令集体系结构包括定义各种**状态单元**、**指令集**和它们的**编码**、一组**编程规范**和**异常事件处理**。

### 4.1.1程序员可见的状态

程序员可见的状态为Y86-64程序中的指令**可以读取或修改的处理器状态的部分**。Y86-64的状态类似于x86-64，有15个**程序寄存器**：%rax、%rcx、%rdx、%rbx、%rsp、%rbp、%rsi、%rdi和%r8到%r14。每个程序寄存器存储一个64位的字，寄存器%rsp作为栈指针。此外，有3个一位的**条件码**：ZF、SF和OF，它们保存着**最近的算术或逻辑指令**所造成影响的有关信息。**程序计数器（PC）**存放当前正在执行指令的地址。**内存**从概念上来说是一个很大的字节数组，保存着程序和数据，Y86-64程序使用**虚拟地址**来引用内存位置，硬件和操作系统软件联合起来会将虚拟地址**翻译**成**实际物理地址**。程序状态的最后一个部分是**状态码Stat**，它表明程序执行的总体状态，会指示是**正常运行**，还是出现了**某种异常**，例如当一条指令试图去读非法的内存地址时。

### 4.1.2Y86-64指令及编码

Y86-64指令集基本上是x86-64指令集的一个**子集**，它只包含**8字节**整数操作，寻址方式较少，操作也较少。指令集的一个重要性质就是**字节编码必须有唯一的解释**。这个性质保证了处理器可以**无二义性**地执行目标代码程序。下图为Y86-64指令集的示例。




> 值得注意的细节有以下几点：

> - 在地址计算中，Y86-64不支持**变址寄存器**和任何寄存器值的**伸缩**，同x86-64一样，也不允许从一个内存地址直接传送到另一个内存地址，另外，也不允许将立即数传送到内存。
> - *halt*指令将停止指令的执行。x86-64中有一个与之相当的指令*hlt*，但x86-64的应用程序不允许使用这条指令，因为它会导致整个系统**暂停运行**。对于Y86-64来说，执行*halt*指令会导致处理器停止，并将状态码设置为**HLT**。

可以看到，Y86-64指令集的每条指令需要1~10个字节不等，这取决于需要哪些字段。一条指令含有一个单字节的**指令指示符**，可能含有一个单字节的**寄存器指示符**，还可能含有一个8字节的**常数字**。每条指令的第一个字节表明**指令的类型**。这个字节分为两个部分，高4位是**代码**部分，低4位（$f_n$）是**功能**部分:

| 指令 | 编码 | 指令 | 编码 |
|---|---|---|---|
| addq | 60 | jge | 75 |
| subq | 61 | jg | 76 |
| andq | 62 | rrmovq | 20 |
| xorq | 63 | cmovle | 21 |
| jmp | 70 | cmovl | 22 |
| jle | 71 | cmove | 23 |
| jl | 72 | cmovne | 24 |
| je | 73 | cmovge | 25 |
| jne | 74 | cmovg | 26 |

注意到，*rrmovq*与条件传送有同样的指令代码，可以把它看成一个**无条件传送**。

在Y86-64指令集中，15个程序寄存器按照顺序每个都有一个相对应的范围在0到0xE之间的**寄存器标识符**。程序寄存器存在CPU中的一个**寄存器文件**中，这个**寄存器文件**就是一个小的、以寄存器ID作为地址的**随机访问存储器**。而在这里的硬件设计中，当需要指明不应访问任何寄存器时，就用ID值0xF来表示。比如对于哪些只需要一个操作数的指令（pushq、popq等），就会将另一个寄存器指示符设置为0xF。为了描述的简单性，处理器使用的是**绝对寻址方式**，并且，所有整数采用**小端法**编码。

例如，用十六进制来表示指令`rmmovq %rsp,0x123456789abcd(%rdx)`的字节编码。我们知道`rmmovq`的第一个字节为40，源寄存器%rsp应该编码放在$r_A$字段中，基址寄存器%rdx应该编码放在$r_B$字段中。最后，偏移量编码放在8字节的常数字中(注意为小端法，因此这里需要**按字节反序**）。因此指令的编码为0x4042cdab896745230100。

---

# 7.链接

**链接**是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个过程可被**加载**（复制）到内存并**执行**。**链接**可以执行于**编译**时，也就是在源代码被翻译为机器代码时；也可以执行于**加载**时，也就是在程序被**加载器**加载到内存时；甚至可以执行于**运行**时，也就是由**应用程序**来执行。现代系统中，**链接**是由叫做**链接器**的程序自动执行的。

链接的重要作用在于它们使得**分离编译**成为可能，任一个大型应用的各个模块可以被独立地**修改**和**编译**。

# 7.1编译器驱动程序

我们常用地GCC就是一种**编译器驱动程序**，它会调用**语言预处理器**、**编译器**、**汇编器**和**链接器**，来讲程序从**ASCII码源文件**翻译成**可执行文件**。例如，若我们在命令行中执行这条指令：

```
linux> gcc -Og -o prog main.c sum.c
```

那么驱动程序的行为如下：

1. 驱动程序首先运行**C预处理器（cpp）**，它将C的源程序*main.c*翻译成一个ASCII码的中间文件*main.i*：`cpp [other arguments] main.c /tmp/main.i`。
2. 接下来，驱动程序运行**C编译器（ccl）**，它将*main.i*翻译成一个ASCII汇编语言文件*main.s*：`ccl /tmp/main.i -Og [ither arguments] -o /tmp/main.s`。
3. 然后，驱动程序运行**汇编器（as）**，它将*main.s*翻译成一个**可重定位**目标文件*main.o*：`as [other arguments] -o /tmp/main.o /tmp/main.s`。驱动程序经过同样的过程生成*sum.o*。
4. 最后，驱动程序运行**链接器（ld）**，它将*main.o*和*sum.o*以及一些必要的系统目标文件组合起来，创建一个**可执行**目标文件*prog*：`ld -o prog [system object files and args] /tmp/main.o /tmp/sum.o`。
5. 在运行可执行文件*prog*时，*shell*会调用操作系统中一个叫做**加载器**的函数，它将可执行文件*prog*中的代码和数据复制到内存，然后将**控制**转移到这个程序的开头。

# 7.2静态链接

目标文件有三种形式：

- **可重定位目标文件**：包含二进制代码和数据，其形式可以在编译时与其他可重定位目标文件合并，创建一个可执行目标文件。
- **可执行目标文件**：包含二进制代码和数据，其形式可以直接被**复制到内存**并**执行**。
- **共享目标文件**：一种特殊类型的可重定位目标文件，可以在加载或者运行时被**动态地**加载进内存并链接。

**静态链接器**以一组**可重定位目标文件**和**命令行参数**作为输入，生成一个**完全链接的**、**可以加载和运行的**可执行目标文件作为输出。输入的**可重定位目标文件**由各种不同的**代码和数据节**组成，每一节都是一个**连续的**字节序列，**指令**在一节中，**初始化了的全局变量**在另一节中，而**未初始化的变量**又在另外一节中。

为了构造可执行文件，链接器必须完成两个主要任务：

- **符号解析**：链接器需要将每个符号**引用**正好和一个符号**定义**关联起来。
- **重定位**：编译器和汇编器生成从地址0开始的代码和数据节。链接器通过把每个符号定义与一个内存位置关联起来，从而**重定位**这些节，然后修改所有对这些符号的**引用**，使得它们指向这个内存位置。

# 7.3可重定位目标文件

典型的**ELF**可重定位目标文件具有两个部分，分别为**节**和**节头部表**。前者由一个16字节的**ELF头**开始，这个序列描述了**生成该文件的系统的字的大小和字节顺序**。**ELF头**剩下的部分包含帮助链接器**语法分析和解释**目标文件的信息。**不同节的位置和大小**是由**节头部表**描述的。夹在ELF头和节头部表之间的都是**节**。一个典型的ELF可重定位目标文件包含下面几个节：

- **.text**：已编译程序的机器代码。
- **.rodata**：**只读数据**，比如开关语句的跳转表。
- **.data**：**已初始化**的**全局和静态**C变量。局部C变量在运行时被保存在栈中，不出现在*.data*和*.bss*节中。

- **.bss**：**未初始化**的全局和静态C变量，以及所有**被初始化为0**的全局和静态变量。在目标文件中这个节**不占据实际空间**，而仅仅是一个**占位符**。目标文件中，**未初始化变量**不需要占据任何实际的磁盘空间。
- **.symtab**：一个符号表，它存放在程序中**定义**和**引用**的**函数**和**全局变量**的信息。
- **.rel.text**：一个*.text*节中位置的列表。当链接器把这个目标文件和其他文件组合时，需要**修改**这些位置。一般而言，任何**调用外部函数**或者**引用全局变量的指令都需要修改**，而调用本地函数的指令则不需要修改。
- **.rel.data**：被**模块引用**或**定义**的所有**全局变量**的重定位信息。一般而言，任何已初始化的全局变量，如果其初始值时一个全局变量地址或者外部定义函数的地址，都需要被修改。
- **.debug**：一个调试符号表，其条目是程序中定义的**局部变量**和**类型定义**，程序中定义和引用的**全局变量**，以及原始的**C源文件**。只有以`-g`选项调用编译器驱动程序才会得到这张表。
- **.line**：原始C源程序中的**行号**和*.text*节中机器指令的映射。同样，只有以`-g`选项调用编译器驱动程序才会得到这张表。
- **.strtab**：一个字符串表。其内容包括*.symtab*和*.debug*节中的符号表，以及节头部中的节名字。字符串表就是以*null*结尾的字符串的序列。

## 7.4符号和符号表

每个**可重定位目标模块**m都有一个包含m**定义和引用的符号的信息的**符号表。其中有三种不同的符号：

- 由模块m定义并能被其他模块引用的**全局符号**。全局链接器符号对应于**非静态的C函数和全局变量**。
- 由其他模块定义并被模块m引用的**全局符号**。这些符号成为**外部符号**，对应于在其他模块中定义的**非静态C函数**和**全局变量**。
- 只被模块m定义和引用的**局部符号**。它们对应于带*static*属性的C函数和全局变量。这些符号在模块m中可见，但是不能被其他模块引用。

值得注意的是，**本地链接器符号**和**本地程序变量**是不同的。例如，*.symtab*中的符号表就不包含对应于本地非静态程序变量的任何符号，这些符号在运行时在栈中被管理。此外，定义为*static*的本地过程变量是不在栈中管理的。

> 在C中，源文件扮演模块的角色。任何带有**static**属性声明的全局变量或者函数都是模块**私有的**。类似地，任何不带**static**属性声明地全局变量和函数都是**公共的**，可以被其他模块访问。

**符号表**是由**汇编器**构造的，使用编译器输出到汇编语言*.s*文件中的符号。符号表包含一个**条目**的数组。每个条目大致包含以下几个部分：

```
typedef struct{
    int name;       //String table offset(字符表中的字节偏移)
    char type:4;    //Function or data(4 bytes)
        binding:4;  //Local or Global(4 bytes)
    char reserved;  //Unused
    short section;  //Section header index
    long value;     //Section offset or absolute address
    long size;      //Object size in bytes
}ELF64_symbol;
```

注意，对于可重定位的模块来说，*value*是距定义目标的节的**起始位置**的偏移；而对于可执行目标文件来说，该值是一个**绝对运行时地址**。*section*表示的是每个符号被分配到目标文件的哪一节，ELF用一个**整数索引**来表示各个节。有三个特殊的**伪节**（它们在节头部表中是没有条目的）：**UNDEF**代表未定义的符号，也就是在本目标模块中引用，却在其他地方定义的符号；**ABS**代表不该被重定位的符号；**COMMON**表示还未被分配位置的未初始化的数据目标。

# 7.5符号解析

链接器解析符号引用的方法是**将每个引用与它输入的可重定位目标文件的符号表中的一个确定的符号定义关联起来。**

对那些**引用**和**定义**在相同模块中的**局部符号**的引用，符号解析非常简单，编译器只允许每个模块中每个局部符号有一个定义。但是，对**全局符号**的引用解析就棘手很多。当编译器遇到一个不是在当前模块中定义的符号（变量或函数名），会**假定**该符号是在其他某个模块中定义的，生成一个**链接器符号表条目**，并把它交给**链接器**处理（如果链接器在它的任何输入模块中都找不到这个被引用的符号定义，就输出一条错误信息并终止）。此外，多个目标文件可能会定义相同名字的全局符号，在这种情况下，链接器要么标志一个错误，要么以某种方法选出一个定义并抛弃其他定义。

> 对于C++和Java中的**重载函数**，链接器之所以可以区分是因为编译器将每个唯一的**方法和参数列表**组合编码成一个对链接器来说唯一的名字。

## 7.5.1对多重定义的全局符号的解析

定义函数和已初始化的全局变量为**强符号**，定义未初始化的全局变量为**弱符号**，Linux链接器使用如下几条规则来处理**多重定义**的符号名：

1. 不允许有多个同名的**强符号。**
2. 如果有一个**强符号**和多个**弱符号**同名，那么选择**强符号。**
3. 如果有多个**弱符号**同名，那么从这些**弱符号**中任意选择一个。

> 此前提到过，编译器会按照一定规则来把符号分配为COMMON和.bss。实际上，采用这个惯例是因为当编译器遇到某个**弱全局符号**时，它并不知道其他模块是否也定义了x，如果是，它无法预测链接器该使用x的多重定义中的哪一个。因此，编译器把x分配成COMMON，将决定权留给链接器。

## 7.5.2与静态库的链接

在7.5.1节中，我们假设链接器读取一组可重定位目标文件，并把它们链接起来。实际上，所有的编译系统都提供一种机制，将所有相关的目标模块打包成为一个单独的文件，称为**静态库**，它可以用作链接器的输入。当链接器构造一个输出的**可执行文件**时，它**只复制**静态库里**被应用程序引用的**目标模块。

相关的函数可以被编译为独立的目标模块，然后封装成一个单独的**静态库文件。**然后，应用程序可以通过在命令行上指定单独的文件名字来使用这些在库中定义的函数。在Linux系统中，静态库以一种称为**存档**的特殊文件格式存放在**磁盘**中。存档文件是一组连接起来的**可重定位目标文件**的集合，有一个头部用来描述每个成员目标文件的大小和位置。存档文件名由后缀*.a*标识。例如，使用C标准库和数学库中函数的程序可以用形式如下的命令行来编译和链接（实际上，C驱动器总是传送*libc.a*给链接器）：

```
linux> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

## 7.5.3使用静态库来解析引用

在符号解析阶段，链接器**从左到右**按照它们在编译器驱动程序命令行上出现的顺序来扫描**可重定位目标文件**和**存档文件。**（驱动程序自动将命令行上的所有*.c*文件翻译成*.o*文件。）在这次扫描中，链接器维护一个可重定位目标文件中的集合**E**（这个集合中的文件会被**合并**起来形成可执行文件），一个**未解析的符号集合**（即引用了但是尚未定义的符号）**U**，以及一个在前面输入文件中**已定义**的符号集合**D。**初始时，U、E、D集合均为空，链接的步骤如下：

- 对于命令行上的每个输入文件$f$，链接器会判断$f$是一个目标文件还是一个存档文件。如果$f$是一个**目标文件**，那么链接器把$f$添加到E，修改U和D来反映$f$中的符号定义和引用，并继续下一个输入文件。
- 如果$f$是一个**存档文件**，那么链接器就尝试匹配U中**未解析的符号**和由**存档文件成员定义的符号。**如果某个存档文件成员m，定义了一个符号来解析U中的一个引用，那么就将m加到E中，并且链接

器修改U和D来反映m中的符号定义和引用。最后，任何不包含在E中的成员目标文件都被丢弃，而链接器将继续处理下一个文件。

- 当链接器完成了对命令行上输入文件的扫描后，如果**U是非空的**，那么链接器就会输出一个错误并终止。否则，它会合并和重定位E中的目标文件，构建输出的可执行文件。

容易发现，上面的算法产生的结果与命令行上文件的顺序密切相关。命令行中，如果定义一个符号的库出现在引用这个符号的目标文件之前，那么这个引用就不能被解析，链接失败。因此，**关于库的一般准则是将它们放在命令行的末尾**，如果库之间不是相互独立的，就需要对它们进行排序，使得对于每个存档文件的成员外部引用的符号s，在命令行中**至少有一个s的定义是在对s的引用之后的**。当然，如果需要满足依赖需求，也可以在命令行上重复库。

> 注，由于目标文件一定会被添加到E中，在将静态库都放在命令行末尾的前提下，只需考虑静态库之间的依赖关系而不需考虑静态库对某个目标文件的依赖。

# 7.6重定位

当链接器完成了**符号解析**后，就将代码中的每个符号引用和**正好一个**符号定义关联起来。此时，链接器知道它的输入目标模块中的**代码节**和**数据节**的确切大小，就可以开始**重定位**了。**重定位**由两个部分组成，分别为**重定位节和符号定义**以及**重定位节中的符号引用**。前者会将所有相同类型的节合并为同一类型的新的**聚合节**，并将运行时内存地址赋给新的聚合节，赋给输入模块中定义的每个节，以及赋给输入模块定义的每个符号。后者则会修改代码节和数据节中对每个符号的引用，使得它们指向正确的**运行时地址**。

## 7.6.1重定位条目

当汇编器生成一个目标模块时，它并不知道数据和代码最终将放在内存中的什么位置，它也不知道这个模块引用的任何外部定义的函数或者全局变量的位置。所以，无论何时汇编器遇到对最终位置未知的目标引用，它就会生成一个**重定位条目**，告诉链接器在将目标文件合并成可执行文件时应该如何修改这个引用。代码的重定位条目放在*.rel.text*中，已初始化数据的重定位条目则放在*.rel.data*中。以下为ELF重定位条目的格式：

```
typedef struct {
    long offset;        //需要被修改的引用的节偏移
    long type:32,       //告知链接器如何修改新的引用
         symbol:32;     //标识出被修改引用应该指向的符号
    long addend;        //一个有符号常数，某些类型的重定位要使用它对被修改引用的值做偏移调整
} Elf64_Rela;
```

ELF定义了32种不同的**重定位类型**，两种最基本的重定位类型如下：

- **R_X86_64_PC32**：重定位一个使用32位**PC相对地址**的引用。一个PC相对地址就是距程序计数器的当前运行时值的**偏移量**。
- **R_X86_64_32**：重定位一个使用32位**绝对地址**的引用。通过绝对寻址，CPU直接使用在指令中编码的32位值作为有效地址。

这两种重定位类型支持**x86-64小型代码模型**，该模型假设可执行目标文件中的代码和数据的总体大小小于2GB，因此在运行时可以使用32位PC相对地址来访问。

## 7.6.2 重定位符号引用

对于C代码*main.c*：

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

通过**OBJDUMP**对GCC汇编产生的*main.o*进行反汇编，结果如下：

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi
                        a: R_X86-64_32 array
   e:   48 8d 0d 00 00 00 00    lea    0x0(%rip),%rcx
                        f: R_X86_64_PC32 sum-0x4
  13:   48 83 c4 08             add    $0x08,%rsp
  17:   c3                      retq
```

可以看到，*main*函数引用了两个全局符号：*array*和*sum*，对每个引用，汇编器产生一个**重定位条目**，显示在引用的后面一行上。这些重定位条目告诉链接器对*sum*的引用要使用**32位PC相对地址**进行重定位，而对array的引用要使用**32位绝对地址**进行重定位。

#### 7.6.2.1重定位PC相对引用

链接器对*sum*的引用为**重定位PC相对引用**。可以看到，相对应的*call*指令开始于节偏移0xe的地方，包括1字节的操作码*0xe8*，后面（0xf）跟着的 便是对目标*sum*的32位PC相对引用的**占位符**。相应的重定位条目r由4个字段组成：

```
r.offset = 0xf
r.symbol = sum
r.type   = R_X86_64_PC32
r.addend = -4
```

这些字段会告诉链接器修改**开始于偏移量0xf**处的**32位PC相对引用**，让其在运行时指向*sum*例程。假设链接器已经确定$ADDR(main) = ADDR(.text) = 0x4004d0$，以及$ADDR(r.symbol) = ADDR(sum) = 0x4004e8$，那么链接器会首先计算出引用的运行时地址：

$$\begin{aligned} refaddr &= ADDR(s) \ + \ r.offset \\ &= 0x4004d0 \ + \ 0xf \\ &= 0x4004df \end{aligned}$$

然后，更新该引用，使得它在运行时指向*sum*程序：

$$\begin{aligned} *refptr &= (unsigned)(ARRD(r.symbol) - refaddr + r.addend) \\ &= (unsigned)(0x4004e8) - 0x4004df + (-4)) \\ &= (unsigned)(0x5) \end{aligned}$$

运行到*call*指令时，CPU首先将**PC**压入栈中，再给它加上上面算出的0x5。注意到，由于执行完*call*指令跳转时PC计数器的值为紧跟在*call*指令后的那条指令的地址，因此需要加上$r.addend$，也就是给计算出来的引用再减去$0x4$，这样才能跳到我们想要跳到的*sum*例程的第一条指令。

#### 7.6.2.2重定位绝对引用

在上面反汇编代码的第四行中，*mov*指令将*array*的地址（1个32位立即数值）复制到寄存器%edi中。*mov*指令开始于节偏移量0x9的位置，包括1字节的操作码0xbf，后面（0xa）跟着的便是对*array*的32位绝对引用的**占位符**。相应的重定位条目r如下：

```
r.offset = 0xa
r.symbol = array
r.type   = R_X86_64_32
r.addend = 0
```

这样，链接器会修改从偏移量0xa开始的绝对引用，在运行时它将会指向array的第一个字节。假设编译器已经确定$ARRD(r.symbol) = ADDR(array) = 0x601018$，则会更新引用为：

$$\begin{aligned} *refptr &= (unsigned)(ARRD(r.symbol) + r.addend) \\ &= (unsigned)(0x601018 + 0) \\ &= (unsigned)(0x601018) \end{aligned}$$

## 7.7可执行目标文件

可执行目标文件的格式类似于可重定位目标文件的格式。ELF头描述文件的总体格式，它还包括程序的**入口点**，也就是当程序运行时要执行的第一条指令的地址。*.text*、*.rodata*、*.data*节与可重定位目标文件中的节很相似，不过这些节已经被**重定位**到它们**最终的运行时内存地址**。

当我们在命令行中输入`linux> prog`时，由于`prog`不是一个内置的*shell*命令，所以*shell*会认为*prog*是一个可执行目标文件，通过调用某个驻留在存储器中称为**加载器**的操作系统代码来运行它。加载器将可执行目标文件中的代码和数据从磁盘中复制到内存中，然后通过跳转到程序的**入口点**来运行该程序。这个将程序复制到内存并运行的过程就叫做**加载**。

> ELF可执行文件被设计得很容易加载到内存。可执行文件的**连续的片**被映射到连续的内存段。**程序头部表**描述了这种**映射关系**，从程序头部表中，可以看到会根据可执行文件的内容来初始化两个内存段，分别具有**读/执行访问权限**以及**读/写访问权限**。前者用来保存ELF头、程序头部表以及*.init*、*.text*、*.rodata*节，后者则主要用来保存*.data*和*.bss*节。

在加载器跳转到程序的**入口点**之后，也就是*_start*函数的地址。这个函数是在系统目标文件*ctrl.o*中定义的。*_start*函数调用**系统启动函数** *__libc_start_main*，该函数定义在*libc.so*中。它初始化执行环境，调用用户层的*main*函数，处理*main*函数的返回值，并且在需要的时候把控制返回给**内核**。

> 所谓**内核**，就是指操作系统驻留在内存中的部分。

## 7.8动态链接共享库

7.5节中讨论的**静态库**解决了许多关于如何让大量相关函数对应用程序可用的问题，但是，其仍有一些明显的缺点，必须其需要定期维护与更新，程序员在每次使用时都需要了解到更新情况并显式地重新连接。另一个问题是，比如几乎每个C程序都会使用标准I/O函数`printf()`和`scanf()`，如果对于所有运行的进程，都将这些函数的代码复制到内存中，那么是一种极大的浪费。

**共享库**是致力于解决静态库缺陷的一个现代创新产物，它是一个目标模块，在运行或加载时，可以加载到**任意的**内存地址，并和一个在内存中的程序连接起来。这个过程称为**动态链接**，是有一个叫做**动态链接器**的程序来执行的。**共享库**也称为**共享目标**，在linux系统中通常用*.so*后缀来表示。微软的操作系统也大量地使用了共享库，它们称为**DLL**。其基本思路是当创建可执行文件时，静态执行一些链接，然后在程序加载时，**动态完成**连接过程。

# 7.9位置无关代码

可以加载而无需重定位的代码称为**位置无关代码（PIC）**。用户对GCC使用*-fpic*选项指示GNU编译系统生成PIC代码。共享库的编译必须总是使用该选项。

- PIC数据引用：**编译器利用了这样一个有用的事实：无论在内存中的何处加载一个目标模块，数据段与代码段的距离总是保持不变，也就是说，代码段中任何指令和数据段中任何变量之间的**距离**都是一个**运行时常量，**与代码段和数据段的**绝对内存位置**无关。

  编译器会首先在数据段开始的地方创建一个**全局偏移量表（GOT）**，在GOT中，每个被这个目标模块引用的**全局数据目标**都有一个8字节条目。在加载时，动态链接器会**重定位**GOT中的每个条目，使得它包含目标的正确的**绝对地址。**由此，编译器利用代码段和数据段之间的不变距离，产生**PC相对引用。**

- PIC函数调用：**当程序调用一个由共享库定义的函数时，由于定义它的共享模块在运行时可以被加载到任意位置，因此编译器没有办法预测这个函数的**运行时地址。**GNU编译系统由此使用了一种称为**延迟绑定**的技术，将**过程地址的绑定推迟到第一次调用该过程时**。

  **延迟绑定**是通过两个数据结构**GOT**和**过程链接表（PLT）**之间的交互来实现的。如果一个目标模块调用定义在共享库中的任何函数，那么它就有自己的GOT和PLT。且，GOT是数据段的一部分，PLT是代码段的一部分。

  > **过程连接表（PLT）**是一个数组，其中每个条目是**16字节代码**，每个条目都负责调用一个具体的函数。PLT[1]调用系统启动函数，其初始化执行环境并调用*main*函数。从PLT[2]开始的条目调用用户代码调用的函数。
  >
  > 和PLT联合使用时，GOT[0]和GOT[1]包含**动态链接器**在解析函数地址时会使用的信息。GOT[2]是动态链接器在*ld-linux.so*模块中的入口点。其余的每个条目都对应于一个被调用的函数，其地址需要**在运行时被解析**。

  GOT和PLT**协同工作**的步骤如下（假设**第一次**调用函数*addvec*）：

  1. 不直接调用*addvec*，程序调用进入PLT[2]，这是*addvec*的PLT条目。
  2. 第一条PLT指令通过GOT[4]进行**间接跳转**，这是因为**每个GOT条目初始时都指向它对应的PLT条目的第二条指令**，这个简介跳转知识简单地把控制传送回**PLT[2]中的下一条指令。**
  3. 在把*addvec*的ID（0x1）压入栈中之后，PLT[2]跳转到PLT[0]。
  4. PLT[0]通过GOT[1]间接地把动态链接器地一个参数压入栈中，然后通过GOT[2]简介跳转进动态链接器中。动态链接器使用两个栈条目来确定*addvec*的**运行时位置**，用这个地址重写GOT[4]，在把控制传递给*addvec*。

  虽然第一次调用时开销很大，但是之后的每次调用中，由于*addvec*的运行时地址已经被写到了GOT[4]中，因此过程简单地在第二步中便可以跳转到正确的位置。

# 7.10库打桩机制

Linux链接器支持一个很强大的技术称为**库打桩**，它允许你截获对**共享库函数**的调用，取而代之执行自己的代码。使用打桩机制，你可以追踪对某个特殊库函数的调用次数，验证和追踪它的输入和输出值，甚至把它替换成一个**完全不同**的实现。基本思想如下：给定一个需要**打桩**的**目标函数**，创建一个**包装函数**，它的原型与目标函数完全一样。使用某种特殊的**打桩机制**，你局可以**欺骗**系统调用包装函数而不是目标函数了。包装函数通常会执行它自己的逻辑，然后调用目标函数，再将目标函数的返回值传递给调用者。

**打桩**可以发生在**编译时**、**连接时**或**程序被加载和执行时。**