静态链接之重定位

上文初步介绍了静态链接符号解析的内容，着重介绍了可重定位目标文件中符号表的内容和多重定义符号解析的内容。一个必须明确的事实是：每个可重定位文件内地址都是从0开始的，即使是运行在实地址模式下，依旧需要给每个段加上段偏移才是最终运行时各指令和数据的地址。

现代操作系统运行时各指令和数据的地址实际上是通过虚拟地址来完成取指取数据工作的，所以我们在重定位工作上主要是完成以下三个工作：一是将用于链接的多个可执行目标文件的节聚合形成聚合节，二是结合操作系统虚拟存储器及进程运行时存储器映像将各个节的起始位置改成具体值（不再是从0开始），三是将机器码中所有的符号引用结合寻址方式和实际定义的数据或函数的地址，改写涉及符号引用部分的机器指令中偏移地址部分。

首先假设我们已经完成了链接形成了我们所需的可执行文件，此时当我们在命令行下通过输入./可执行文件名即可通过操作系统提供的加载器来加载可执行文件到内存中，然后修改PC的值，使PC的值变成所加载程序的入口点的地址（entry point）来运行程序。

为了讲清楚这个问题，首先需要了解运行时程序的存储器映像（Linux）。

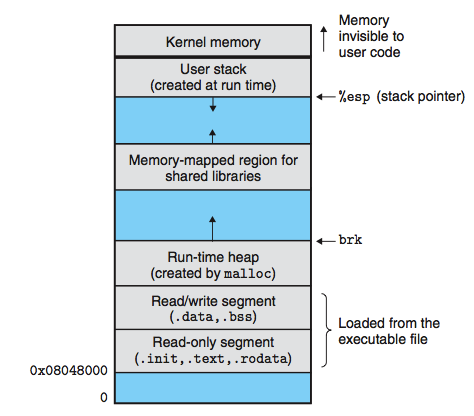


图1：Linux运行时存储器映像

进程相对于程序最大的区别在于进程是程序动态运行的实体，而不仅仅是存储在外存的二进制指令和数据。学习过现代操作系统虚拟存储器的人都知道，每一个进程在被操作系统创建时都会为每个进程创建独立的页表，页表保存了虚拟存储器中各页（虚页）和物理内存中映射关系，所以不同的两个进程是可以有完全相同的地址的。为了说清楚这个抽象的描述我们举个例子。

进程A和进程B都被加载到内存中并且都在就绪队列中等待调度，操作系统调度进程A先执行指令，此时PC变成进程A的入口指令的地址，假设地址是VA，VA并不是指令所在内存中的地址，此时需要通过查看页表将VA->PA得到指令在内存中的实际地址，之后取指令并执行。即使此时进程B被调度且入口指令地址依旧是VA，但是由于进程B有独立的页表，所以在VA从虚拟地址专为物理地址时会被映射到不同于VA的地址。这种方式保证了各个进程都有完整的且连续的4G（32位虚存）线性地址空间，这个机制实现了同一进程代码、数据可以离散地分配在物理地址空间，大大提高了物理内存使用率。另外，这种方式对于编程人员而言，得到了更大的空间（虚拟地址空间），虽然并不是实际的内存空间，但有了连续的线性地址之后就可以更好地安排程序各个部分的位置了。

所以无论其他结构如何，每一个进程对于操作系统来说，它们的进程存储器映像都是相同的。这听起来还是有些抽象，简单来说就是可执行文件作为二进制文件是存储在磁盘上了，当我们创建进程时为每个进程创建页表，创建页表的过程就是把可执行文件的各个段映射成图1的存储模型中，比如将可执行文件的.text节映射到虚地址空间的VA处，此时可以得到一个虚拟页号VN（VA的高位部分）。之后取得内存中空闲物理页地址PA，同样可以得到一个物理页号PN（PA的高位部分），然后在页表项中建立起虚拟页号到物理页号的映射关系VN->PN。

实际上加载程序的过程仅仅加载了程序的头部涉及定位信息的部分，指令和代码此时都还在磁盘上，当程序被操作系统从就绪队列中调度取得CPU控制权之后，当取该进程的指令时会因为请求的页不在内存而产生缺页中断，然后通过请求调页机制将对应的内容从磁盘复制到内存中之前页表中映射到的物理页中。之后等进程再次被调度时候就能取得相应的指令或数据了。

上面大段描述只是为了能大致说清楚虚拟存储器，当然，上面有一个过程是和接下来要说的链接是密切相关的：存储器映射。我们将可执行文件映射到具体的虚拟存储器地址空间中的这个过程并不是一个随意的行为，映射必须遵照图1中各段位置来完成。

假设是4G虚拟地址空间，虚地址最高的1G地址空间是分配给操作系统映射代码和内核堆栈的，当然这不在本文讨论范文内。虚拟地址空间[0, 0x08048000 - 1]是不被分配的，紧接着该位置的映射可执行文件.init、.text和.rodata节的内容，在往上紧接着就是用来映射.bss和.data节的地址空间。在这部分之上是留给运行时程序动态分配内存的堆区的映射空间，所以堆的地址是向上增长的。接下来剩下的虚拟地址空间将会分配给共享库以及程序的栈区，值得一提的是栈去是紧贴着内核空间的，它从高地址向低地址增长。

因为程序在运行时的地址并不是从0开始的，大致位置可以推断出应该比0x080480000稍微大一些，这点可以从图1的.text段的位置运行时存储器映像看得出。所以在可执行文件中，我们也需要对各个段的起始位置重新定位，而各个可重定位目标文件的地址都是从0开始的，即使多个目标文件聚合之后依旧是从0地址开始的。

所以第一件事就是给每个段加上相应的偏移地址，这就解决了一个问题，进程的运行时地址是什么时候形成的？显然是在链接阶段。

现在脑海里有这个为了到虚拟地址的映射而加上的偏移量的印象就可以了。接下来我们将先讨论可重定位目标文件聚合问题。

在聚合过程中，链接器将输入的各个模块同一类型的节聚合形成一个新的聚合节，比如main.o中.text节和swap.o中的.text节形成最终可执行文件的.text节。接着，链接器将运行时地址赋给各个聚合节，这里的运行时地址就是前面所说的可执行文件的各部分映射到程序运行时存储映像中的地址，所以各节的地址往往很大并且这个值并不是最终在内存中的实际地址。

在完成聚合和给各节赋运行时地址之后，每一个在编译阶段分配存储空间的变量和函数的指令都有了唯一的运行时地址。这时距离我们最后的目标已经很接近了，当前已经有了可执行文件的雏形，拥有完整的节结构，每个变量函数都可以通过唯一的地址进行索引和访问。但是现在还有个问题就是，之前各模块间相互引用的变量和函数还没建立起联系来，要使各最终成为一个整体还要修改.text节和.data节中所有对符号的引用，让其指向真正的定义部分的运行时地址。

这一步的完成需要之前可重定位目标模块中.rel.data节和.rel.text节记录的信息来完成。在生成目标模块阶段，编译器并不知道最终的运行时地址是多少，也并不知道所引用的符号被定义在哪个模块，它的地址又是多少。所以汇编器在读入汇编代码时，一旦读到符号引用就会生成一个可重定位条目，如果是已初始化变量的引用放在.rel.data节中，如果是函数名引用放在.rel.text节中。

下图为readelf读取的main.o和swap.o两个目标文件的重定位节.rel.text和.rel.data内容。

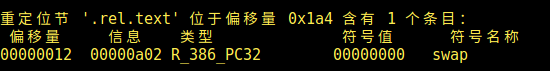


图2：main.o重定位节

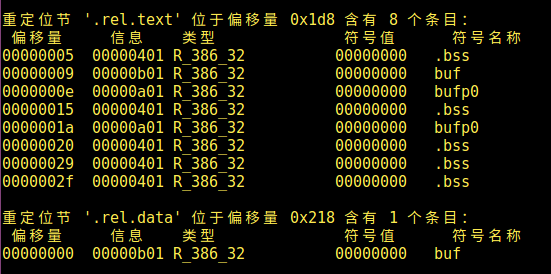


图3：swap.o重定位节

重定位节我们所需要掌握的三个属性是：符号名称、类型以及偏移量。符号名称之前说符号表时已经说的很明白了，类型列指的是当指令引用了某个符号的变量或函数要通过什么寻址方式去找到它，偏移指的是符号出现的位置相对于引用它的.text节首地址的偏移量（后面反汇编时会说明）。这里只说两中最常见的方式：R\_386\_PC32，R\_386\_32，其中R表示重定位，386表示处理器类型。所以R\_386\_PC32表示机器指令中若涉及对该符号的引用时，采取什么寻址方式找到引用符号的具体定义。在图2中可见，当main函数引用swap函数时，call指令将通过相对寻址方式进行寻址，相对寻址就是实际数据定义的运行时地址相对于当前PC值的偏移量。不过需要注意的是，当CPU取得call指令之后，PC会自动加上call执行的长度而存储下一条指令的地址，而实际我们所需的填入机器指令中的偏移量是相对于call这条指令的，所以在计算时需要先减去PC值减去call指令长度，这个值与数据地址的差值填入指令偏移量中。相对寻址通常用于函数调用。

类型为R\_386\_32相对会简单很多，重定位使用引用内容的绝对地址，指令编码中直接给出32位的数据地址，然后在执行指令时直接到该地址取得数据。绝对寻址通常属于全局数据寻址。

阐述这个问题前，还需要明确两点：一是只有.text节的指令部分需要根据寻址方式来修改指令中偏移地址，其他节不需要改变内容，二是聚合完成之后，聚合的.text节中包含了多个原本独立的.text节内容，所以聚合.text由多个独立的代码段组成。

下面的伪代码展示了重定位时修改指令偏移地址字段过程。

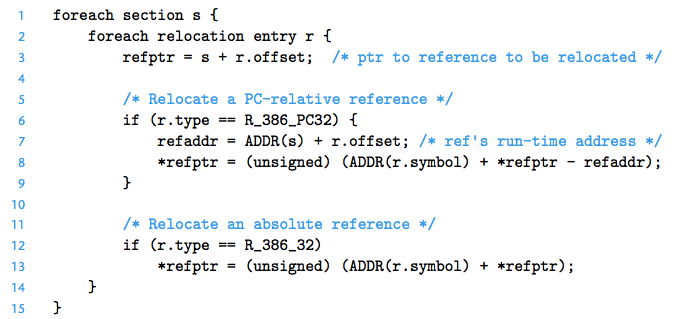


图4：重定位算法

对于聚合的.text中各个子节.text（后面用s表示），在与各节s相关联的重定位条目r上进行迭代。我们假设各个子节的首地址为s，refptr用子节首地址加上重定位表中符号引用出现时的偏移量（则这里swap出现在<main +0x12位置上>），所以refptr指向含有符号引用指令的偏移地址字段的地址。

这里需要停下来解释一下上面伪代码中各变量的意义。ADDR(s)和ADDR(r.symbol)是链接器为节s和r的符号的运行时地址。所以refaddr = ADDR(s) + r.offset表示的意义是：有符号引用指令的偏移地址字段的程序运行时地址。比较refaddr和refptr可见二者均是指向符号引用的偏移地址字段，但是区别在于refaddr是运行时该字段的地址，refptr是当前静态的二进制文件下的地址。这里二者的差值refaddr – refptr正是存储器映射时的偏移量。

找到了要改写的偏移字段的地址，接下来就要根据寻址类型来生成填入偏移字段的值了。如果是相对寻址方式的话（R\_386\_PC32），通过ADDR(r.symbol) – refaddr + \*refptr来得到，ADDR(r.symbol) – refaddr可以得到符号出现的运行时地址和待填入的偏移地址字段之间的差值。又因为是相对PC的寻址方式，此时PC指向下一跳指令，所以这里要进行一些处理，减掉该指令原先偏移字段的值，至于为什么这么减后面反汇编代码之后会再分析。通过这步生成的偏移值写入引用了符号的那条指令的偏移地址字段。

这样在程序加载之后，按序执行.text节中的机器指令，如果遇到符号引用，就根据既定的寻址方式和偏移字段的值生成实际定义的变量或函数的地址。

下图是通过objdump –d生成的反汇编指令的节选。

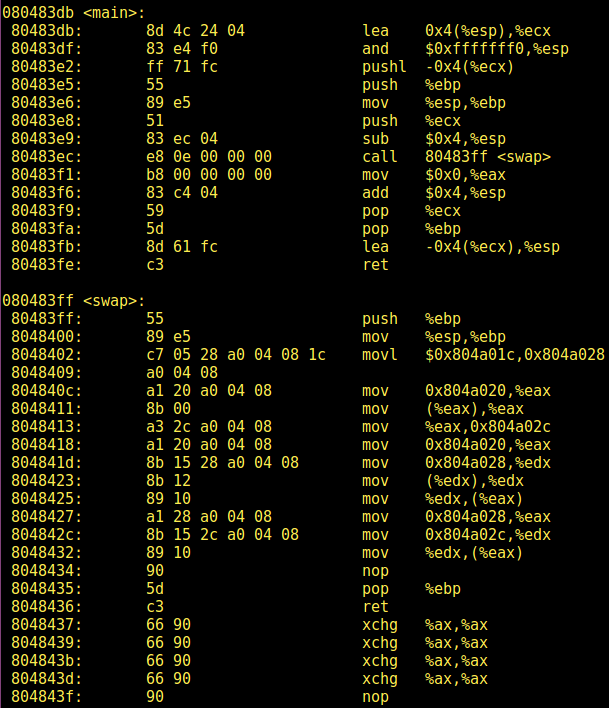


图5：反汇编可执行文件

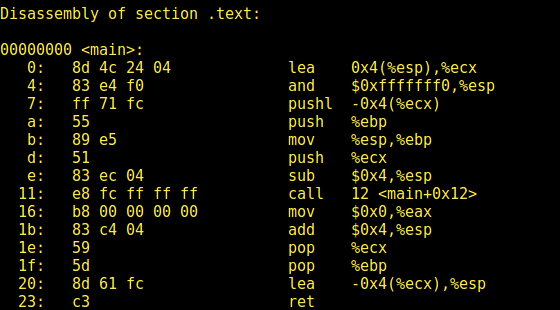


图6：反汇编main.o

图6中在尚未完成链接前，汇编代码中偏移为11的位置上是跳转指令call，call的指令占了5个字节，指令字段占1个字节（0xe8表示call指令），偏移字段占4个字节。偏移字段从左到右分别是0xfc ff ff ff ，因为我的计算机是小端序的，所以实际这个值是0xfffffffc，对应的十进制数是-4。

所以上面的算法中ADDR(r.symbol) – refaddr + refptr，原本计算完ADDR(r.symbol) – refaddr之后得到偏移量应该再减去call指令的5个字节，但是因为这个我们需要的仅仅是写入偏移字段，所以实际上call指令字段占的1个字节是不用计算的，只需要减去4个字节，这里refptr指针指向的值(\*refptr)正好为-4，这样就可以计算出真实的写入位置（refptr指针所指地址）和真实的偏移量大小了（引用符号地址距离定义数据的真实位置）。

我们就图5的例子分析一下这个过程，先列出三条接下来需要用到的指令的地址：

main中call引用时符号的地址：0x080483ec + 1 = 0x080483ed

call下一条指令地址：0x080483f1

swap函数入口地址：0x080483ff

首先是swap入口相距call指令地址的偏移量：0x080483ff – 0x080483ed = 0x12，按照上面分析，0x12还要减4，得到0xe。最后把0xe转换成32为补码写入call指令偏移字段，从图5可见偏移字段确实是0x0000000e。

当然了，在计算时我们还可以通过更加简单的方式得到这个结果，虽然计算机没有办法直接采用这种方式。就是直接用swap的入口地址减去call下一条指令地址即可，0x080483ff – 0x080483f1 = 0xe。

到这里基本算是完成重定位过程了，接下来看一下可执行文件的ELF格式和如何将可执行文件映射到虚拟存储器的各个段中。

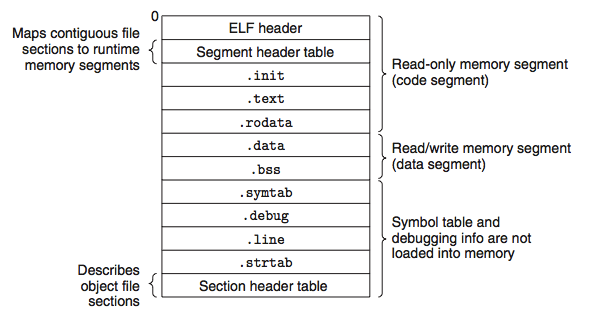


图7：可执行文件ELF格式

重定位处理完成之后形成的可执行文件的基本构造如上，ELF头相比可重定位目标文件的ELF头多了个程序入口地址，也就是程序执行的第一条指令地址。多出的.init节中定义了初始化代码，其他几个节和可重定位目标文件相比几乎没有什么区别，只不过在可执行文件中每个节由多个相同类型的节聚合而成。.rel.data和.rel.text节因为完成了重定位工作，在可执行文件中已经没有了。从这个时刻开始，原本叫做节的部分统一叫做段。

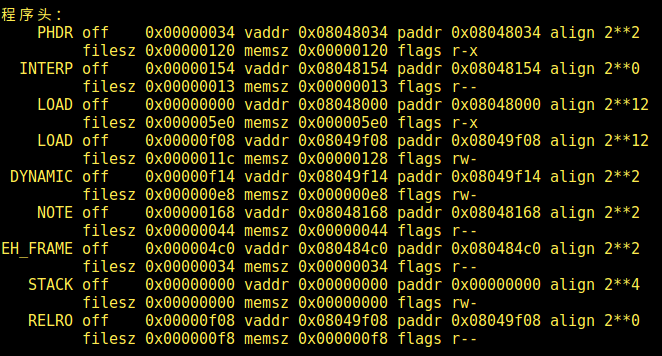


图8：段头部表

想要查看加载信息可以通过objdump –x查看，可以得到可执行文件被映射到了虚拟存储器的具体什么位置及其他基本信息。只关注以LOAD开头的数据，其它部分不在本文讨论范围内。

其中off表示文件偏移，vaddr表示映射到的虚拟地址，align表示段对齐的单位，filesz表示目标文件中段大小，flags表示该段的读、写、执行权限。

结合前面提到每个进程都把.text段映射到0x08048000位置，可以推知这里第一个LOAD是把.text、.rodata、.init（均为只读段）段映射到虚拟地址0x08048000，该只读段大小为0x5e0字节，对齐大小为4KB（一个基本页的大小），运行时对该区域数据的权限只有读和执行（r-x）。也就是说，当链接后形成的可执行文件是明确知道将会被映射到虚拟存储器什么位置以及每一条指令和数据的运行时地址的。

所谓的创建进程不过是根据可执行文件的说明书，为进程创建页表，其中页表的虚页号可以通过映射关系得到。

到这里静态链接重定位的讨论告一段落。