原文链接:http://www-128.ibm.com/developerworks/cn/linux/l-dynlink/

程序的链接和装入及Linux下动态链接的实现

2003 年 8 月 10 日

程序的链接和装入存在着多种方法，而如今最为流行的当属动态链接、动态装入方法。本文首先回顾了链接器和装入器的基本工作原理及这一技术的发展历史，然后 通过实际的例子剖析了Linux系统下动态链接的实现。了解底层关键技术的实现细节对系统分析和设计人员无疑是必须的，尤其当我们在面对实时系统，需要对 程序执行时的时空效率有着精确的度量和把握时，这种知识更显重要。

链接器和装入器的基本工作原理

一个程序要想在内存中运行，除了编译之外还要经过链接和装入这两个步骤。从程序员的角度来看，引入这两个步骤带来的好处就是可以直接在程序中使用 printf和errno这种有意义的函数名和变量名，而不用明确指明printf和errno在标准C库中的地址。当然，为了将程序员从早期直接使用地 址编程的梦魇中解救出来，编译器和汇编器在这当中做出了革命性的贡献。编译器和汇编器的出现使得程序员可以在程序中使用更具意义的符号来为函数和变量命 名，这样使得程序在正确性和可读性等方面都得到了极大的提高。但是随着C语言这种支持分别编译的程序设计语言的流行，一个完整的程序往往被分割为若干个独 立的部分并行开发，而各个模块间通过函数接口或全局变量进行通讯。这就带来了一个问题，编译器只能在一个模块内部完成符号名到地址的转换工作，不同模块间 的符号解析由谁来做呢？比如前面所举的例子，调用printf的用户程序和实现了printf的标准C库显然就是两个不同的模块。实际上，这个工作是由链接器来完成的。

为了解决不同模块间的链接问题，链接器主要有两个工作要做——符号解析和重定位:

符号解析: 当一个模块使用了在该模块中没有定义过的函数或全局变量时，编译器生成的符号表会标记出所有这样的函数或全局变量，而链接器的责任就是要到别的模块中去查找它们的定义，如果没有找到合适的定义或者找到的合适的定义不唯一，符号解析都无法正常完成。

重定位: 编译器在编译生成目标文件时，通常都使用从零开始的相对地址。然而，在链接过程中，链接器将从一个指定 的地址开始，根据输入的目标文件的顺序以段为单位将它们一个接一个的拼装起来。除了目标文件的拼装之外，在重定位的过程中还完成了两个任务: 一是生成最终 的符号表; 二是对代码段中的某些位置进行修改，所有需要修改的位置都由编译器生成的重定位表指出。

举个简单的例子，上面的概念对读者来说就一目了然了。假如我们有一个程序由两部分构成，m.c中的main函数调用f.c中实现的函数sum:

```
/* m.c */
int i = 1;
int j = 2;
extern int sum();
void main()
{
        int s;
        s = sum(i, j);
/* f.c */
int sum(int i, int j)
{
        return i + j;
}
```

在Linux用gcc分别将两段源程序编译成目标文件:

```
$ gcc -c m.c
$ gcc -c f.c
```

我们通过objdump来看看在编译过程中生成的符号表和重定位表:

```
$ objdump -x m.o
……
SYMBOL TABLE:
……
00000000 g             0 .data  00000004 i
00000004 g             0 .data  00000004 j
00000000 g             F .text  00000021 main
00000000         *UND*  00000000 sum
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000007 R_386_32          j
0000000d R_386_32          i
00000013 R_386_PC32        sum
```

首先，我们注意到符号表里面的sum被标记为UND（undefined），也就是在m.o中没有定义，所以将来要通过ld（Linux下的链接器）的符 号解析功能到别的模块中去查找是否存在函数sum的定义。另外，在重定位表中有三条记录，指出了在重定位过程中代码段中三处需要修改的位置，分别位于7、 d和13。下面以一种更加直观的方式来看一下这三个位置:

```
$ objdump -dx m.o
Disassembly of section .text:
00000000 <main>:
   0:   55                                              push   %ebp
   1:   89 e5                                           mov    %esp,%ebp
   3:   83 ec 04                                        sub    $0x4,%esp
   6:   a1 00 00 00 00                          mov    0x0,%eax
7: R_386_32     j
   b:   50                                              push   %eax
   c:   a1 00 00 00 00                          mov    0x0,%eax
d: R_386_32     i
  11:   50                                              push   %eax
  12:   e8 fc ff ff ff                          call   13 <main+0x13>
13: R_386_PC32  sum
  17:   83 c4 08                                        add    $0x8,%esp
  1a:   89 c0                                           mov    %eax,%eax
  1c:   89 45 fc                                        mov    %eax,0xfffffffc(%ebp)
  1f:   c9                                              leave
  20:   c3                                              ret
```

以sum为例，对函数sum的调用是通过call指令实现的，使用IP相对寻址方式。可以看到，在目标文件m.o中，call指令位于从零开始的相对地址 12的位置，这里存放的e8是call的操作码，而从13开始的4个字节存放着sum相对call的下一条指令add的偏移。显然，在链接之前这个偏移量 是不知道的，所以将来要来修改13这里的代码。那现在这里为什么存放着0xfffffffc（注意Intel的CPU使用little endian的编址方式）呢？这大概是出于安全的考虑，因为0xfffffffc正是－4的补码表示（读者可以在gdb中使用p /x -4查看），而call指令本身占用了5个字节，因此无论如何call指令中的偏移量不可能是－4。我们再看看重定位之后call指令中的这个偏移量被修 改成了什么:

```
$ gcc m.o f.o
$ objdump -dj .text a.out | less
Disassembly of section .text:
……
080482c4 <main>:
……
80482d6:       e8 0d 00 00 00          call   80482e8 <sum>
80482db:       83 c4 08                       add    $0x8,%esp
……
080482e8 <sum>:
……
```

可以看到经过重定位之后，call指令中的偏移量修改成0x0000000d了，简单的计算告诉我们：0x080482e8－0x80482db=0xd。这样，经过重定位之后最终的可执行程序就生成了。

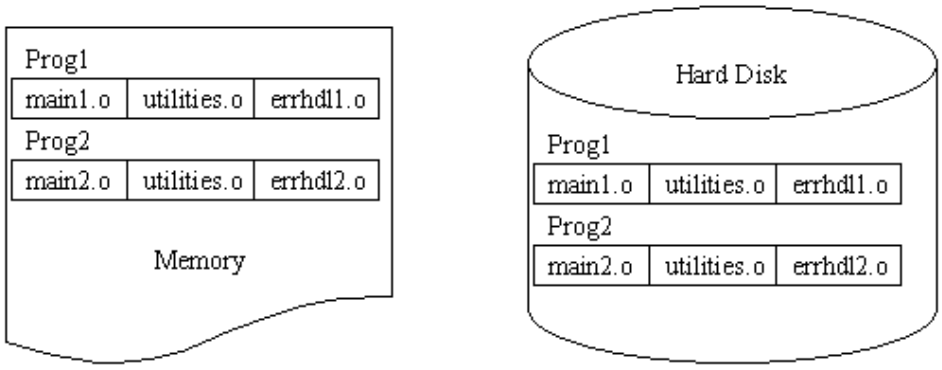
可执行程序生成后，下一步就是将其装入内存运行。Linux下的编译器（C语言）是cc1，汇编器是as，链接 器是ld，但是并没有一个实际的程序对应装入器这个概念。实际上，将可执行程序装入内存运行的功能是由execve(2)这一系统调用实现的。简单来讲， 程序的装入主要包含以下几个步骤：

- 读入可执行文件的头部信息以确定其文件格式及地址空间的大小；
- 以段的形式划分地址空间；
- 将可执行程序读入地址空间中的各个段，建立虚实地址间的映射关系；
- 将bbs段清零；
- 创建堆栈段；
- 建立程序参数、环境变量等程序运行过程中所需的信息；
- 启动运行。

链接和装入技术的发展史

一个程序要想装入内存运行必然要先经过编译、链接和装入这三个阶段，虽然是这样一个大家听起来耳熟能详的概念，在操作系统发展的过程中却已经经历了多次重大变革。简单来讲，可以将其划分为以下三个阶段：

1．静态链接、静态装入
这种方法最早被采用，其特点是简单，不需要操作系统提供任何额外的支持。像C这样的编程语言从很早开始就已经支持分别编译了，程序的不同模块可以并行开 发，然后独立编译为相应的目标文件。在得到了所有的目标文件后，静态链接、静态装入的做法是将所有目标文件链接成一个可执行映象，随后在创建进程时将该可 执行映象一次全部装入内存。举个简单的例子，假设我们开发了两个程序Prog1和Prog2，Prog1由main1.c、utilities.c以及errhdl1.c三部分组成，分别对应程序的主框架、一些公用的辅助函数（其作用相当于库）以及错误处理部分，这三部分代码编译后分别得到各自对应的目 标文件main1.o、utilities.o以及errhdl1.o。同样，Prog2由main2.c、utilities.c以及 errhdl2.c三部分组成，三部分代码编译后分别得到各自对应的目标文件main2.o、utilities.o以及errhdl2.o。值得注意的 是，这里Prog1和Prog2使用了相同的公用辅助函数utilities.o。当我们采用静态链接、静态装入的方法，同时运行这两个程序时内存和硬盘 的使用情况如图1所示：



可以看到，首先就硬盘的使用来讲，虽然两个程序共享使用了utilities，但这并没有在硬盘保存的可执行程序映象上体现出来。相 反，utilities.o被链接进了每一个用到它的程序的可执行映象。内存的使用也是如此，操作系统在创建进程时将程序的可执行映象一次全部装入内存， 之后进程才能开始运行。如前所述，采用这种方法使得操作系统的实现变得非常简单，但其缺点也是显而易见的。首先，既然两个程序使用的是相同的 utilities.o，那么我们只要在硬盘上保存utilities.o的一份拷贝应该就足够了；另外，假如程序在运行过程中没有出现任何错误，那么错 误处理部分的代码就不应该被装入内存。因此静态链接、静态装入的方法不但浪费了硬盘空间，同时也浪费了内存空间。由于早期系统的内存资源十分宝贵，所以后 者对早期的系统来讲更加致命。
2．静态链接、动态装入

既然采用静态链接、静态装入的方法弊大于利，我们来看看人们是如何解决这一问题的。由于内存紧张的问题在早期的系统中显得更加突出，因此人们首先想 到的是要解决内存使用效率不高这一问题，于是便提出了动态装入的思想。其想法是非常简单的，即一个函数只有当它被调用时，其所在的模块才会被装入内存。所 有的模块都以一种可重定位的装入格式存放在磁盘上。首先，主程序被装入内存并开始运行。当一个模块需要调用另一个模块中的函数时，首先要检查含有被调用函 数的模块是否已装入内存。如果该模块尚未被装入内存，那么将由负责重定位

的链接装入器将该模块装入内存，同时更新此程序的地址表以反应这一变化。之后，控 制便转移到了新装入的模块中被调用的函数那里。

动态装入的优点在于永远不会装入一个使用不到的模块。如果程序中存在着大量像出错处理函数这种用于处理小概率事件的代码，使用这种方法无疑是卓有成效的。在这种情况下，即使整个程序可能很大，但是实际用到（因此被装入到内存中）的部分实际上可能非常小。

仍然以上面提到的两个程序Prog1和Prog2为例，假如Prog1运行过程中出现了错误而Prog2在运行过程中没有出现任何错误。当我们采用静态链接、动态装入的方法，同时运行这两个程序时内存和硬盘的使用情况如图2所示:

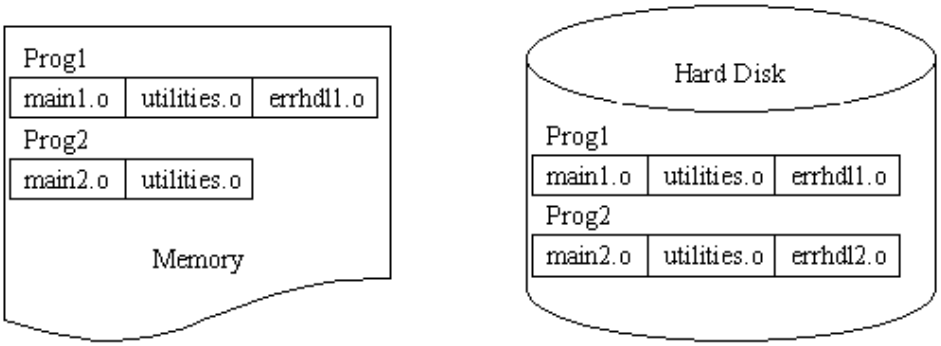


图 **2**采用静态链接、动态装入方法，同时运行**Prog1**和**Prog2**时内存和硬盘的使用情况

可以看到，当程序中存在着大量像错误处理这样使用概率很小的模块时，采用静态链接、动态装入的方法在内存的使用效率上就体现出了相当大的优势。到此为止，人们已经向理想的目标迈进了一部，但是问题还没有完全解决——内存的使用效率提高了，硬盘呢?

3. 动态链接、动态装入

采用静态链接、动态装入的方法后看似只剩下硬盘空间使用效率不高的问题了，实际上内存使用效率不高的问题仍然没有完全解决。图2中，既然两个程序用 到的是相同的utilities.o，那么理想的情况是系统中只保存一份utilities.o的拷贝，无论是在内存中还是在硬盘上，于是人们想到了动态 链接。

在使用动态链接时，需要在程序映象中每个调用库函数的地方打一个桩（stub）。stub是一小段代码，用于定位已装入内存的相应的库；如果所需的库还不在内存中，stub将指出如何将该函数所在的库装入内存。

当执行到这样一个stub时，首先检查所需的函数是否已位于内存中。如果所需函数尚不在内存中，则首先需要将其装入。不论怎样，stub最终将被调用函数的地址替换掉。这样，在下次运行同一个代码段时，同样的库函数就能直接得以运行，从而省掉了动态链接的额外开 销。由此，用到同一个库的所有进程在运行时使用的都是这个库的同一份拷贝。

下面我们就来看看上面提到的两个程序Prog1和Prog2在采用动态链接、动态装入的方法，同时运行这两个程序时内存和硬盘的使用情况（见图3）。仍然假设Prog1运行过程中出现了错误而Prog2在运行过程中没有出现任何错误。

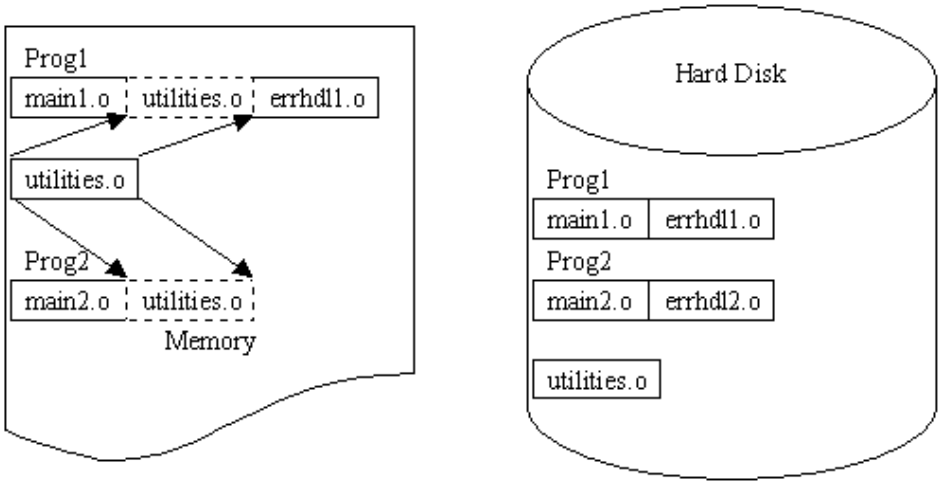


图 **3**采用动态链接、动态装入方法，同时运行**Prog1**和**Prog2**时内存和硬盘的使用情况

图中，无论是硬盘还是内存中都只存在一份utilities.o的拷贝。内存中，两个进程通过将地址映射到相同的utilities.o实现对其的共享。 动态链接的这一特性对于库的升级（比如错误的修正）是至关重要的。当一个库升级到一个新版本时，所有用到这个库的程序将自动使用新的版本。如果不使用动态 链接技术，那么所有这些程序都需要被重新链接才能得以访问新版的库。为了避免程序意外使用到一些不兼容的新版的库，通常在程序和库中都包含各自的版本信 息。内存中可能会同时存在着一个库的几个版本，但是每个程序可以通过版本信息来决定它到底应该使用哪一个。如果对库只做了微小的改动，库的版本号将保持不 变；如果改动较大，则相应递增版本号。因此，如果新版库中含有与早期不兼容的改动，只有那些使用新版库进行编译的程序才会受到影响，而在新版库安装之前进 行过链接的程序将继续使用以前的库。这样的系统被称作共享库系统。

Linux下动态链接的实现

如今我们在Linux下编程用到的库（像libc、QT等等）大多都同时提供了动态链接库和静态链接库两个版本的库，而gcc在编译链接时如果不加 -static选项则默认使用系统中的动态链接库。对于动态链接库的原理大多数的书本上只是进行了泛泛的介绍，在此笔者将通过在实际系统中反汇编出的代码 向读者展示这一技术在Linux下的实现。

下面是个最简单的C程序hello.c:

```
#include <stdio.h>
int main()
{
    printf("Hello, world\n");
    return 0;
}
```

在Linux下我们可以使用gcc将其编译成可执行文件a.out:

```
$ gcc hello.c
```

程序里用到了printf，它位于标准C库中，如果在用gcc编译时不加-static的话，默认是使用libc.so，也就是动态链接的标准C库。在gdb中可以看到编译后printf对应如下代码 ：

```
$ gdb -q a.out
(gdb) disassemble printf
Dump of assembler code for function printf:
0x8048310 <printf>:     jmp    *0x80495a4
0x8048316 <printf+6>:   push   $0x18
0x804831b <printf+11>:  jmp    0x80482d0 <_init+48>
```

这也就是通常在书本上以及前面提到的打桩（stub）过程，显然这并不是真正的printf函数。这段stub代码的作用在于到libc.so中去查找真正的printf。

```
(gdb) x /w 0x80495a4
0x80495a4 <_GLOBAL_OFFSET_TABLE_+24>:    0x08048316
```

可以看到0x80495a4处存放的0x08048316正是pushl $0x18这条指令的地址，所以第一条jmp指令没有起到任何作用，其作用就像空操作指令nop一样。当然这是在我们第一次调用printf时，其真正的 作用是在今后再次调用printf时体现出来的。第二条jmp指令的目的地址是plt，也就是procedure linkage table，其内容可以通过objdump命令查看，我们感兴趣的就是下面这两条对程序的控制流有影响的指令

```
$ objdump -dx a.out
……
080482d0 >.plt>:
 80482d0:       ff 35 90 95 04 08       pushl  0x8049590
 80482d6:       ff 25 94 95 04 08       jmp    *0x8049594
……
```

第一条push指令将got（global offset table）中与printf相关的表项地址压入堆栈，之后jmp到内存单元0x8049594中所存放的地址0x4000a960处。这里需要注意的一 点是，在查看got之前必须先将程序a.out启动运行，否则通过gdb中的x命令在0x8049594处看到的结果是不正确的。

```
(gdb) b main
Breakpoint 1 at 0x8048406
(gdb) r
Starting program: a.out
Breakpoint 1, 0x08048406 in main ()
(gdb) x /w 0x8049594
0x8049594 <_GLOBAL_OFFSET_TABLE_+8>:    0x4000a960
(gdb) disassemble 0x4000a960
Dump of assembler code for function _dl_runtime_resolve:
0x4000a960 <_dl_runtime_resolve>:       pushl  %eax
0x4000a961 <_dl_runtime_resolve+1>:     pushl  %ecx
0x4000a962 <_dl_runtime_resolve+2>:     pushl  %edx
0x4000a963 <_dl_runtime_resolve+3>:     movl   0x10(%esp,1),%edx
0x4000a967 <_dl_runtime_resolve+7>:     movl   0xc(%esp,1),%eax
0x4000a96b <_dl_runtime_resolve+11>:    call   0x4000a740 <fixup>
0x4000a970 <_dl_runtime_resolve+16>:    popl   %edx
0x4000a971 <_dl_runtime_resolve+17>:    popl   %ecx
0x4000a972 <_dl_runtime_resolve+18>:    xchgl  %eax,(%esp,1)
0x4000a975 <_dl_runtime_resolve+21>:    ret    $0x8
0x4000a978 <_dl_runtime_resolve+24>:    nop
0x4000a979 <_dl_runtime_resolve+25>:    leal   0x0(%esi,1),%esi
End of assembler dump.
```

前面三条push指令执行之后堆栈里面的内容如下:

.....
address of "Hello, world\n"
return address
0x18
0x8049590
Eax
Ecx
Edx

下面将0x18存入edx，0x8049590存入eax，有了这两个参数，fixup就可以找到printf在libc.so中的地址。当fixup返回时，该地址已经保存在了eax中。xchg指令执行完之后堆栈中的内容如下:

.....
address of "Hello, world\n"
return address
0x18
0x8049590
address of printf in libc.so

最妙的要数接下来的ret指令的用法，这里ret实际上被当成了call来使用。ret $0x8之后控制便转移到了真正的printf函数那里，并且清掉了堆栈上的0x18和0x8049584这两个已经没用的参数，这时堆栈便成了下面的样子:

.....
address of "Hello, world\n"
return address

而这正是我们所期望的结果。应该说这里ret的用法与Linux内核启动后通过iret指令实现由内核态切换到用户态的

做法有着异曲同工之妙。很多人都听 说过中断指令int可以实现用户态到内核态这种优先级由低到高的切换，在接受完系统服务后iret指令负责将优先级重新降至用户态的优先级。然而系统启动 时首先是处于内核态高优先级的，Intel i386并没有单独提供一条特殊的指令用于在系统启动完成后降低优先级以运行用户程序。其实这个问题很简单，只要反用iret就可以了，就像这里将ret 当作call使用一样。另外，fixup函数执行完还有一个副作用，就是在got中与printf相关的表项（也就是地址为0x80495a4的内存单 元）中填上查找到的printf函数在动态链接库中的地址。这样当我们再次调用printf函数时，其地址就可以直接从got中得到，从而省去了通过 fixup查找的过程。也就是说got在这里起到了cache的作用。

一点感想

其实有很多东西只要勤于思考，还是能够自己悟出一些道理的。国外有一些高手就是通过能够大家都能见到的的一点点资料，自己摸索出来很多不为人知的秘密。像写《Undocument Dos》和《Undocment Windows》的作者，他就为我们树立了这样的榜样!

学习计算机很关键的一点在于一定要富于探索精神，要让自己做到知其然并知其所以然。侯先生在《STL源码剖析》 一书开篇题记中写到"源码之前，了无秘密"，当然这是在我们手中掌握着源码的情况下，如若不然，不要忘记Linux还为我们提供了大量的像gdb、 objdump这样的实用工具。有了这些得力的助手，即使没有源码，我们一样可以做到"了无秘密"。

参考资料

- John R. Levine.《Linkers & Loaders》.
- 《Executable and Linkable Format》.
- Intel. 《Intel Architecture Software Developer's Manual》. Intel Corporation, 1997.

关于作者

王勇，现在北京航空航天大学计算机学院系统软件实验室攻读计算机硕士学位，主要研究领域为操作系统及分布式文件系统。可以通过 yongwang@buaa.edu.cn与他联系。