



《深入理解计算机系统》 | 链接

唐鱼的学习探索 关注

 0.155 2018.04.01 17:56:21 字数 5,521 阅读 2,724



目 录

链接是将各种不同文件的代码和数据部分收集（符号解析和重定位）起来并组合成一个单一文件的过程。本章节我们将要学习链接器工作的详细原理。通过对这一方面知识的学习，将有助于理解一些危险的编程错误、分离编译的过程、作用域的实现以及如何利用共享库等等。我们将静态链接和动态链接（加载时共享、运行时共享）两个大的方向讲起。废话不多说，开始飙车了。

1.1 编译驱动程序如何工作？

在我的raspberrypi上我创建了两个c程序源文件：main.c和swap.c

```
pi@raspberrypi:~/code $ ls -l
total 8
-rw-r--r-- 1 root root 80 Mar 19 07:44 main.c
-rw-r--r-- 1 root root 158 Mar 19 07:48 swap.c
pi@raspberrypi:~/code $
```

文件目录

文件内容


小鹅通

小鹅通商家数突破100万

— 在线直播课堂系统,最高直减5888元 —

立即注册

广告

唐鱼的学习探索 关注

总资产 29 (约2.85元)

如何高效的准备一次考试

阅读 1,737

都9102年了，你还不知道anki是什么

阅读 102

(a) main.c

```
1  /* main.c */
2  void swap();
3
4  int buf[2] = {1, 2};
5
6  int main()
7  {
8      swap();
9      return 0;
10 }
```

code/link/main.c

(b) swap.c

```
1  /* swap.c */
2  extern int buf[];
3
4  int *bufp0 = &buf[0];
5  int *bufp1;
6
7  void swap()
8  {
9      int temp;
10
11     bufp1 = &buf[1];
12     temp = *bufp0;
13     *bufp0 = *bufp1;
14     *bufp1 = temp;
15 }
```

code/link/swap.c

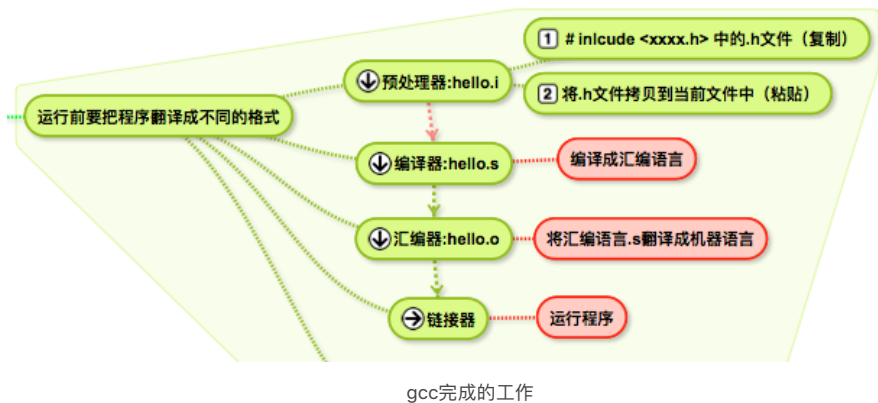
main.c和swap.c

我们通过gcc驱动程序：

```
pi@raspberrypi:~/code $ gcc -g -o p main.c swap.c
pi@raspberrypi:~/code $ ls
main.c  p  swap.c
pi@raspberrypi:~/code $
```

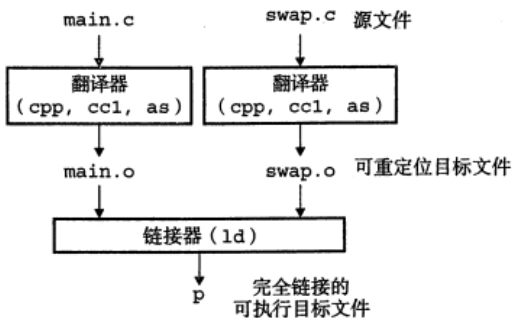
gcc驱动程序

在第一章我们解释过编译驱动程序所完成的工作，如下图：



gcc完成的工作

先是由预处理器 (cpp) 将main.c翻译成中间文件：main.i，接下来是编译器 (cc1) 将main.i翻译成汇编文件main.s。然后是汇编器 (as) 将main.s翻译成一个可重定位的目标文件main.o。最后由链接器 (ld) 将main.o和swap.o以及一些系统目标文件组合起来，创建可执行目标文件p



在以上的这个过程中ld链接器的主要工作：

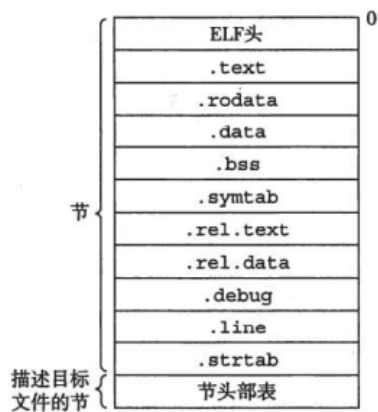
- ① 符号解析。目标文件定义和引用符号，符号解析的目的是将每个符号引用和一个符号定义联系起来；
- ②重定位：把每个符号定义与一个存储器位置联系起来，然后修改对这些符号的引用，是的他们指向这个存储器位置，从而实现重定位。

为了理解这一过程，我们需要补充一些基础知识。

1.2 链接器操作的目标文件究竟是什么？

目标文件一般是由汇编器生成的.o后缀的文件，大概有三种不同的形式：可重定位目标文件；可执行目标文件和共享目标文件。我们接下来讨论的目标文件是基于Unix系统的ELF格式（Exxcutable and Linkable Format），这同Windows系统上的PE（Portable Executable）文件格式在基本概念上其实是相似的：

- ①一个典型的ELF可重定位目标文件的格式：



一个典型的ELF格式可重定位目标文件

- 解释：
- .text：已编译程序的机器码；.rodata：只读数据（read-only-data）；
 - .data：已初始化的全局C变量；.bss：未初始化的全局C变量（better save space）；
 - .symtab：一个符号表（定义和引用的函数和全局变量信息）；
 - .rel.text：代码重定位条目，一个.text节中位置的列表，需要修改的位置；
 - .rel.data: 被模块引用或定义的任何全局变量的重定位信息；
 - .debug：一个调试符号表；.line：原始C源程序中的行号和.text机器指令的映射；
 - .strtab: 一个字符串表

② 符号和符号表（链接器的第一个任务符号解析）

保存于.symtab中的是一个符号表，其是定义和引用函数和全局变量的信息。有三种不同类型的符号：全局符号（不带static），外部引用（external）和本地符号。如果是带有static符号的就会在.data和.bss中为每个定义分配空间，并在.symtab中创建一个唯一名字的本地符号。比如：

```
1  int f()
2  {
3      static int x = 0;
4      return x;
5  }
6
7  int g()
8  {
9      static int x = 1;
10     return x;
11 }
```

中有两个static定义的x变量，其会在.data中分配空间，并在.symtab中创建两个，x.1表示f函数的定义和x.2表示函数g的定义。（注：使用static可以保护你自己的变量和函数）

.symtab符号表的数据结构：

```
code/link/elfstructs.c
1  typedef struct {
2      int name;          /* String table offset */
3      int value;         /* Section offset, or VM address */
4      int size;          /* Object size in bytes */
5      char type:4,       /* Data, func, section, or src file name (4 bits) */
6          binding:4;     /* Local or global (4 bits) */
7      char reserved;     /* Unused */
8      char section;      /* Section header index, ABS, UNDEF, */
9                          /* Or COMMON */
10 } Elf_Symbol;
code/link/elfstructs.c
```

我们给出main.o符号表中的最后三个条目：（开始的都是使用的本地符号）

Num:	Value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	OBJECT	GLOBAL	0	3	buf
9:	0	17	FUNC	GLOBAL	0	1	main
10:	0	0	NOTYPE	GLOBAL	0	UND	swap

我们看到num8处，的全局变量buf定义条目，位于.data(Ndx=3)开始字节偏移为0（value为0）处的8个字节目标（size）。随后是全局符号main的定义，其位于.text(Nex=1)处，偏移字节为0处（value）的17个字节函数。最后一个是swap的引用，所以是Und。

1.3 链接器开始工作了

① 符号解析（开始链接器的第一个任务）

符号解析任务简单的说，就是链接器使得所有模块中的每个符号只有一个定义。链接器在这一个阶段的主要任务就是把代码中的每个符号引用和确定的一个符号定义联系起来。对于本地符号，这个任务相对来说是简单的。复杂的就是全局符号，编译器（cc1）遇到不是在当前模块中

定义的符号时，会假设该符号的定义在其他模块中，生成一个链接器符号交给链接器处理。如果链接器ld在所有的模块中都找不到定义的话就会抛出异常。

这里最容易产生的错误就是当多个模块定义同一个符号的时候，我们的链接器到底怎么做。以C++中的函数重载为例，我们会按照实际的需要重载许多相同名字的函数，链接器（ld）使用一种叫做毁坏的方法（mangling）将相同函数名不同参数的函数，比如foo将会编码成3foo_的形式，实际上还是使得在链接器层面上来看符号是唯一的。

链接器如何解析多重定义的全局符号：

使用如下规则

规则1：不允许多个强符号；

规则2：如果有一个强符号和多个弱符号，那么选择强符号；

规则3：如果有多个弱符号，那么这些弱符号中任意选择一个；

举个例子：链接器试图编译和链接下面两个模块就会参数错误：

```
1  /* foo1.c */           1  /* bar1.c */
2  int main()             2  int main()
3  {                      3  {
4      return 0;           4      return 0;
5  }
```

规则1：不允许多个强符号（两处定义了main）

第二个例子：如果模块中有x未被初始化，链接器会选择定义在另外一个模块中的强符号（这会导致许多不易察觉的错误）

```
1  /* foo3.c */           1  /* bar3.c */
2  #include <stdio.h>      2  int x;
3  void f(void);           3
4                          4  void f()
5  int x = 15213;          5  {
6                          6      x = 15212;
7  int main()              7  }
8  {
9      f();
10     printf("x = %d\n", x);
11     return 0;
12 }
```

会输出x=15212，规则2，函数f将很低调的将x改成15212，对main带来不易察觉的意外！特别是当重复定义的符号有不同的类型时，需要特别的谨慎。编译系统不会发出任何警告，而且会在程序执行很久以后才表现出来。使用GCC-fno-common可以告诉链接器，遇到这类情况，输入一条警告。

如何链接和解析静态库

链接静态库：

像printf等一些常用的函数，都是在libc.a静态库中，静态库以一种存档的特殊文件（.a）格式，将可以定位的目标文件集合成一个.a文件。举一个实际的例子：

我的raspberrypi上创建有这样的文件：

```
pi@raspberrypi:~/code2 $ ls
addvec.c  main2.c  multvec.c  vector.h
```

文件目录

其中：

(a) addvec.o

code/link/addvec.c

```
1 void addvec(int *x, int *y,
2             int *z, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         z[i] = x[i] + y[i];
8 }
```

code/link/addvec.c

(b) multvec.o

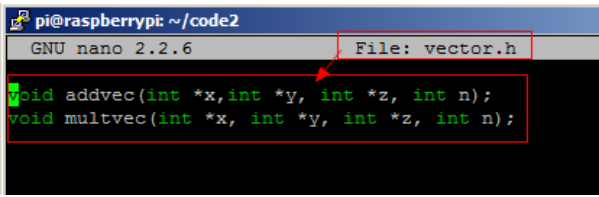
code/link/multvec.c

```
1 void multvec(int *x, int *y,
2              int *z, int n)
3 {
4     int i;
5
6     for (i = 0; i < n; i++)
7         z[i] = x[i] * y[i];
8 }
```

code/link/multvec.c

两个库：addvec.o和multvec.o

我们使用vector.h声明这两个函数：



vector.h

同时使用：main2.c进行函数的调用：

code/link/main2.c

```
1  /* main2.c */
2  #include <stdio.h>
3  #include "vector.h"
4
5  int x[2] = {1, 2};
6  int y[2] = {3, 4};
7  int z[2];
8
9  int main()
10 {
11     addvec(x, y, z, 2);
12     printf("z = [%d %d]\n", z[0], z[1]);
13     return 0;
14 }
```

code/link/main2.c

main2.c

我们现在使用AR工具创建一个静态库：libvector.a文件：

```
pi@raspberrypi: ~/code2
pi@raspberrypi:~/code2 $ gcc -c addvec.c multvec.c
pi@raspberrypi:~/code2 $ ls
addvec.c  addvec.o  main2.c  multvec.c  multvec.o  vector.h
pi@raspberrypi:~/code2 $ ar rcs libvector.a addvec.o multvec.o
pi@raspberrypi:~/code2 $ ls
addvec.c  addvec.o  libvector.a  main2.c  multvec.c  multvec.o  vector.h
pi@raspberrypi:~/code2 $
```

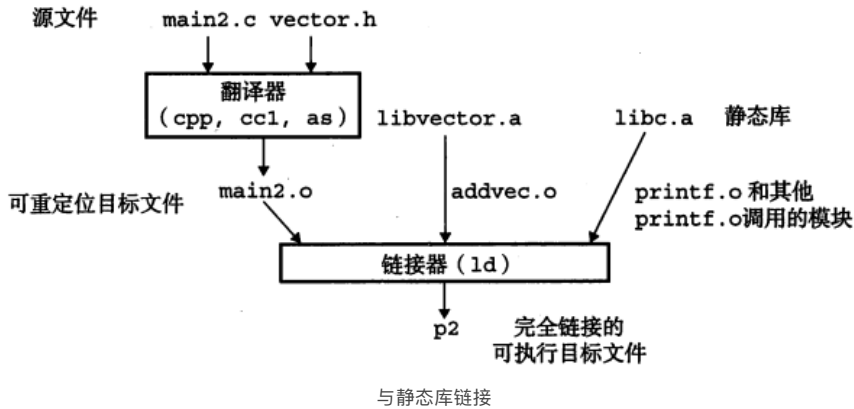
创建libvector.a文件

现在我们使用main2.c函数调用libvector.a库

```
pi@raspberrypi:~/code2 $ gcc -c main2.c
pi@raspberrypi:~/code2 $ ls
addvec.c  libvector.a  main2.o  multvec.o
addvec.o  main2.c  multvec.c  vector.h
pi@raspberrypi:~/code2 $ gcc -static -o p2 main2.o ./libvector.a
pi@raspberrypi:~/code2 $ ls
addvec.c  libvector.a  main2.o  multvec.o  vector.h
addvec.o  main2.c  multvec.c  p2
```

链接静态库libvector.a

这样一个过程可以用下图说明：



解析静态库：解析静态库的过程是按照命令行标识的文件顺序从左到右解析，如果输入文件是一个目标文件(.o)，那么将文件添加到集合E（合并成执行文件）；如果f是一个存档文件（.a），那么就尝试解析集合U（未解析的符号），能够解析的话就将其加载到集合E中去；重复这样的过程直到都解析完毕。

② 重定位

完成了符号解析以后，链接器的第二个任务就是合并输入模块，并为每个符号分配运行时的地址。重定位节和符号定义：在这一步中，链接器将所有模块中的.data节合并成一个文件的.data节，运行时存储器的地址也会赋给新的聚合节。然后就是，重定位节中的符号引用：链接器修改代码节和数据节中对每个符号的引用，使得他们指向正确的运行时地址。这一步要用到重定位条目这一数据结构，我们来描述这个过程：

重定位条目：我们在1.2讲述ELF文件格式的时候说过，.rel.text代表代码重定位条目；.rel.data是已经初始化数据的重定位条目。数据结构如下图：

code/link/elfstructs.c

```

1  typedef struct {
2      int offset;      /* Offset of the reference to relocate */
3      int symbol;      /* Symbol the reference should point to */

```

(注：当汇编器生成一个.o文件模块的时候，它不知道数据和代码最终会放到存储器的什么位置，它只是生成一个重定位条目，放到.rel.text中告诉大家这个内容会在以后修改)

说明：

offset：是需要修改的引用节的偏移；

symbol：标识被修改引用应该指向的符号；

type：告诉连接器如何修改新的引用；

ELF有11种不同的重定位类型：我们只关心常用的两种

R_386_PC32（相对地址引用）和R_386_32（绝对地址引用）

有了重定位的条目，我们也知道了有两种不同的重定位类型，我们下面来看看如何进行符号引用的重新定位：

重定位符号引用：

我们先来看看一段重定位算法的伪代码：

```

1  foreach section s {
2      foreach relocation entry r {
3          refptr = s + r.offset; /* ptr to reference to be relocated */
4
5          /* Relocate a PC-relative reference */
6          if (r.type == R_386_PC32) {
7              refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8              *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
9          }
10
11         /* Relocate an absolute reference */
12         if (r.type == R_386_32)
13             *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
14     }
15 }

```

假设每个节s是一个字节数组，每个重定位条目r是一个Elf32_Rel结构，第三行计算的是需要被重新定位的引用数组s中的地址。然后就根据r的type类型进行不同类型的重定位。上图展示的就是相对地址引用和绝对地址引用两种模式。

例子1：相对地址引用模式（R_386_PC32）

我们回到最开始讲述的main.c和swap.c程序，来看看main.c中的反汇编列表的一个片段：

```

6:  e8 fc ff ff ff      call    7 <main+0x7>  swap();
7:  R_386_PC32 swap     relocation entry

```


这里我们看到call指令开始于字节偏移0x6处的位置，swap函数在main处偏移0x7处的位置。重定位类型使用的是R_386_PC32模式（相对地址引用）。重定位条目的数据结构如下：

```
r.offset = 0x7
r.symbol = swap
r.type   = R_386_PC32
```

Elf32_Rel结构

这个结构告诉我们，修改偏移量为0x7的相对引用，使得它能指向swap程序的位置。假设：两处的地址为：

```
ADDR (s)      = ADDR (.text) = 0x80483b4;
```

```
ADDR(r.symbol) = ADDR(swap) = 0x80483c8;
```

使用refaddr算法计算出引用运行时候的地址为：refaddr = addr (s) + 0x7 = 0x80483bb。然后计算出*refptr：

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr)
          = (unsigned) (0x80483c8 + (-4) - 0x80483bb)
          = (unsigned) (0x9)
```

```
80483ba: e8 09 00 00 00      call    80483c8 <swap>      swap();
```

我们使用的是原值是（-4）经过计算后将修改*refptr为0x9；下面我们来看看置顶到call指令时候的地址情况：

我们看到当前地址外80483ba，CPU执行call指令的时候PC的值是下一条指令的地址80483bf，由于是相对地址引用模式，我们使用计算出来的0x9（*refptr）来重定位执行swap函数的位置：

1. push PC onto stack
2. PC <- PC + 0x9 = 0x80483bf + 0x9 = 0x80483c8

这就是我们之前假设的swap地址的地址。（注：*refptr为什么初始为-4，因为pc总是指向当前指令的下一条指令，不同的机器可能有不同的偏移量）

例子2：绝对地址引用模式（R_386_32）

我们再来示例程序swap.o中int *bufp0 = &buf[0]的情况。反汇编列表如下：

```
00000000 <bufp0>:
0: 00 00 00 00      int *bufp0 = &buf[0];
                   0: R_386_32 buf      Relocation entry
```

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr)
          = (unsigned) (0x8049454 + 0)
          = (unsigned) (0x8049454)
```

由于bufp0是一个已经被初始化的数据目标，在ELF文件结构中位于.data字段位置，反汇编列出的情况表明其位于偏移0x0处且使用R_386_32绝对地址引用模式。现在我们假设地址已经确定是： $\text{addr}(\text{r.symbol}) = \text{addr}(\text{buf}) = 0\text{x}8049454$ 链接器使用我们前面讲过的算法修改引用：

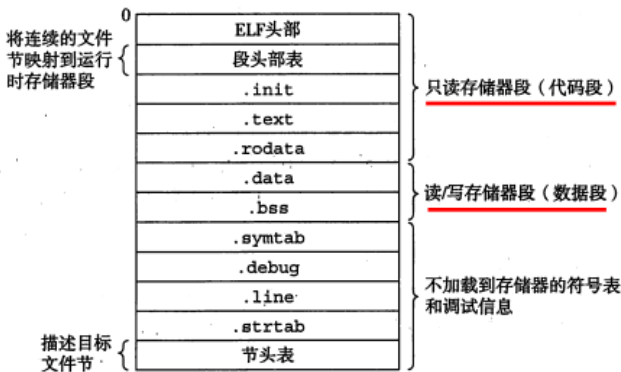
```
0804945c <bufp0>:
804945c: 54 94 04 08                               Relocated!
```

这样就使得refptr直接指向了buf的地址（08049454）也就是如下图所示：

总而言之，链接器绝对在运行时变量bufp0将存放于存储器0x804945c处，并且初始化为0x8049454即buf地址的内容。

1.4 链接器完成工作后生成的目标文件是个什么？

通过前面知识的学习，我们了解到链接器主要完成了两个工作，符号解析和重新定义。将数据和代码合并成为一个可执行的文件，接下来我们看看这个可执行文件的格式是什么，以及如何加载到存储器中开始运行的过程。



① 可执行目标文件格式（一个典型的ELF可执行文件）

说明：

ELF头部：描述文件总体格式，标注出程序入口点；.init：定义了初始化函数；

段头部表：可执行文件是一个连续的片，段头部表中描述了这种映射关系；

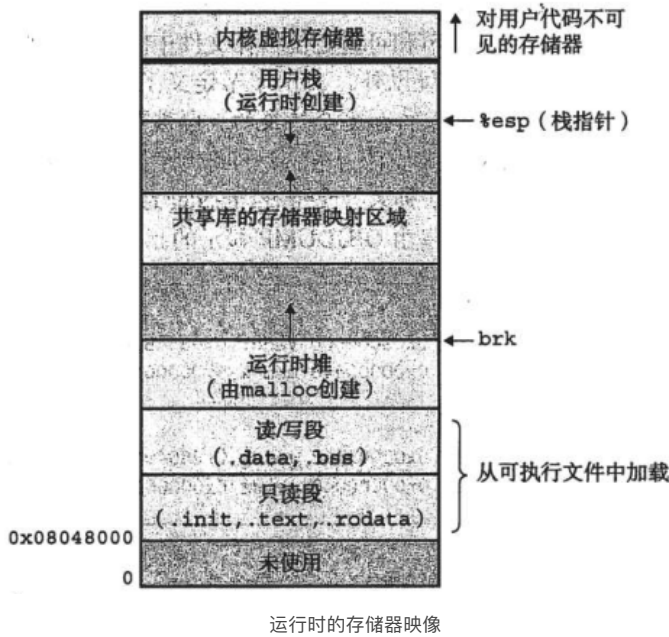
```
pi@raspberrypi:~/code $ gcc -g -o p main.c swap.c
pi@raspberrypi:~/code $ ls
main.c  p  swap.c
pi@raspberrypi:~/code $
```

我们在开始的时候使用main.c和swap.c生成了可执行文件p

我们来看看这个执行文件的反汇编代码：

说明：在段头部表中，我们会看到程序初始化为两个存储器字段，行1和行2是代码段，有读和执行的权限（flags: r-x），开始于存储器地址0x08048000处（vaddr/paddr），该字段大小为0x448（memsz），并且初始化为可执行目标文件的头0x448个字节（filesz）；行3和行4是数据段，有读写的权限（flags），开始于存储器地址：0x08049448处，总大小0x104个字节（memsz），从文件偏移0x448（off）处开始的0xe8（filesz）个字节初始化。

② 如何加载可执行目标文件



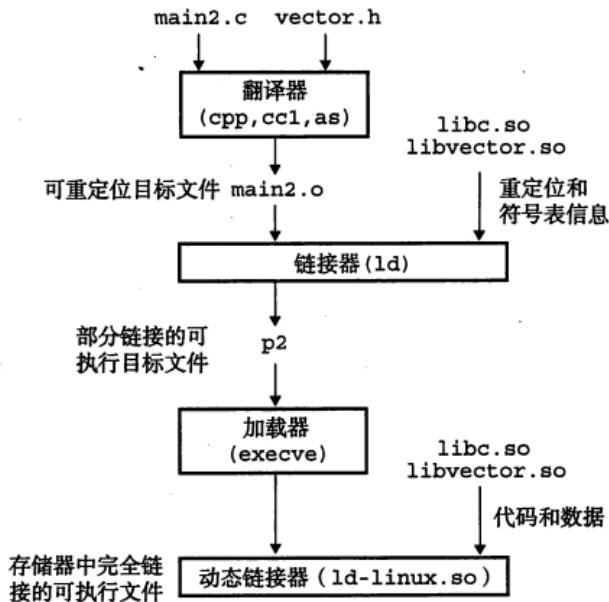
加载后运行的每个Unix程序都有一个镜像，如上图所示。代码段总是从0x08048000开始，数据段是接下来的4kb对齐地址处，运行时堆在读写段之后，使用malloc向上增长；还有一个段为共享库保留。用户栈是在最大合法地址处开始并向下增长。再往上就是不对用户开放的内核虚拟存储器了。

什么是加载？说白了就是将程序拷贝到存储器并运行的过程。这里是由execev函数来调用加载器（驻留在存储器中）完成的，我们要执行p文件的时候，就是使用.p来，加载器就把p的数据和代码拷贝从磁盘拷贝到了存储器中，并通过跳转到ELF头部中的程序入口点开始程序p的执行。

怎样加载？当加载器运行时，就先创建一个存储器映像（上图所示），在ELF可执行文件头部表的指示下，加载器将可执行文件的代码和数据段拷贝到0x0804800处向上的两个段中，然后跳转到程序入口点_start（在ctrl.o中定义）开始执行

1.5 动态链接共享库

① 编译时加载



静态库需要定期的维护和更新，调用的代码还会拷贝到每个运行的进程中去，这是对存储器系统资源的极大浪费。为了弥补这样的缺陷，我们发明了共享库。共享库的一个主要目的就是允许多个正在运行的进程共享存储器中相同的库代码，节约资源。以(.so)结尾的文件，在运行时被加载到任意存储器地址，并和存储器中的程序链接起来，以后的进程要用到这个库就从这个固定的位置开始访问。这一过程的管理交由动态链接器程序来执行。

```
pi@raspberrypi: ~/code2
pi@raspberrypi:~/code2 $ gcc -shared -fPIC -o libvector.so addvec.c multvec.c
pi@raspberrypi:~/code2 $ ls
addvec.c libvector.so main2.c multvec.c vector.h
pi@raspberrypi:~/code2 $
```

我们实际来创建一个.so文件：使用如下方式

说明：-shared指示链接器创建一个共享目标文件；-fPIC生成与位置无关代码

```
pi@raspberrypi:~/code2 $ gcc -o p2 main2.c ./libvector.so
pi@raspberrypi:~/code2 $ ls
addvec.c libvector.so main2.c multvec.c p2 vector.h
pi@raspberrypi:~/code2 $
```

然后创建可执行文件p2：

这个思路很重要：当p2生成的时候没有任何libvector.so的代码和数据被真正拷贝到p2中去，它是在运行的时候与libvector.so链接，p2中只是拷贝了一些重定位和符号表。当加载器加载p2程序开始运行的时候，动态链接器注意到p2中有.interp节，加载器就会加载和运行动态链接器，动态链接器重定位.so的文本和数据到一个存储器段中，然后将p2中的符号引用重新定位到存储器段中已经加载的.so文本和数据的位置。动态链接器完成这些工作以后就会把控制权交给p2，由于共享库(.so)位置固定好了，程序就会开始执行。

② 运行时加载共享库

```
GNU nano 2.2.6      File: dll.c

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2]= {1,2};
int y[2]= {3,4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    handle = dlopen("./libvector.so", RTLD_LAZY);
    if(!handle)
    {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }

    addvec = dlsym(handle, "addvec");
    if((error = dlerror()) != NULL){
        fprintf(stderr, "%s\n",error);
        exit(1);
    }
    addvec(x,y,z,2);
    printf("z=[%d %d]\n", z[0], z[1]);

    if (dlclose(handle) < 0)
    {
```

微软的windows程序开发人员提供共享库来更新软件，通常要求下载最新的dll库，然后在程序下一次执行的时候会自动链接和加载更新后的共享库。我们创建dll.c文件，运行时加载libvector.so

说明：

1>使用dlopen打开本地libvector.so共享库，并解析库中的符号；

2>使用dlsym访问其中的addvec函数，如果存在就返回该函数的地址；

3>使用dlclose卸载共享库；

```
pi@raspberrypi:~/code2 $ gcc -rdynamic -o p3 dll.c -ldl
pi@raspberrypi:~/code2 $ ls
addvec.c  dll.c  libvector.so  main2.c  multvec.c  p2  p3  vector.h
pi@raspberrypi:~/code2 $
```

开始编译运行时共享库：

1.6 与位置无关的代码

我们前面讲过，使用-fPIC（Position-Independent Code）生成与位置无关代码，使得多个进程可以共享相同的库代码。那么多个进程究竟是如何共享程序的一个拷贝库呢？

我们使用的编译库代码，使得这一部分的库代码直接可以加载到存储器中执行，这一过程不需要链接器修改库代码的内容。这样的代码就叫做与位置无关的代码。对于模块内部的调用不需要特殊处理，但是外部定义的函数调用和全局变量的引用就需要链接时重定位。

① PIC数据引用

地址	条目	内容	描述
08049674	GOT[0]	0804969c	.dynamic 节的地址
08049678	GOT[1]	4000a9f8	链接器的标识信息
0804967c	GOT[2]	4000596f	动态链接器中的入口点
08049680	GOT[3]	0804845a	PLT[1] 中 pushl 地址 (printf)
08049684	GOT[4]	0804846a	PLT[2] 中 pushl 地址 (addvec)

当存储器加载一个共享目标模块的时候，数据段总是被分配到紧跟着代码段后，因此任何指令和任何变量之间的距离在运行时都是一个常量。这个很好的特性就被运用起来，编译器在数据段开始地方创建了GOT表（Global Offset Table）（全局偏移量表）如main2.o中的GOT表：

- .dynamic：段的地址，包含动态链接器用来绑定函数地址的信息（符号表、重定位）；
- GOT[1]：定义模块的信息；GOT[2]：延迟绑定代码入口；

```
call L1
L1:  popl %ebx          ebx contains the current PC
      addl $VAROFF, %ebx  ebx points to the GOT entry for var
      movl (%ebx), %eax   reference indirect through the GOT
      movl (%eax), %eax
```

每个被main2.o引用的全局数据对象都有一个条目，编译器还会为每个条目生成一个重定位地址。在加载时动态链接器会重定位到每个正确的地址。我们来看看数据的引用过程：

pop将当前的pc弹出到ebx中，随后的add指令加上一个常量，使得指向正确的变量位置，此处包含了该变量的绝对地址。后面的两条mov指令：第一条eax存放了变量的绝对地址，第二条获

② PIC函数调用

```
PLT[0]
08048444: ff 35 78 96 04 08  pushl 0x8049678  push &GOT[1]
0804844a: ff 25 7c 96 04 08  jmp *0x804967c    jmp to *GOT[2](linker)
08048450: 00 00                                padding
08048452: 00 00                                padding

PLT[1] <printf>
8048454: ff 25 80 96 04 08  jmp *0x8049680    jmp to *GOT[3]
804845a: 68 00 00 00 00 00  pushl $0x0        ID for printf
804845f: e9 e0 ff ff ff    jmp 8048444        jmp to PLT[0]

PLT[2] <addvec>
8048464: ff 25 84 96 04 08  jmp *0x8049684    jump to *GOT[4]
804846a: 68 08 00 00 00 00  pushl $0x8        ID for addvec
804846f: e9 d0 ff ff ff    jmp 8048444        jmp to PLT[0]

<other PLT entries>
```

使用延迟绑定技术，通过GOT与过程链接表PLT（Procedure Linkage Table），将过程的地址绑定延迟到第一次调用该过程时（第一次调用的开销较大）。p2中的PLT表如图：

第一个PLT[0]是一个特殊条目，它跳转到动态链接器中，从PLT[1]开始是每个函数的过程链接表；当addvec被第一次调用的时候（不会立即绑定，延迟一下）将控制传递到PLT[2]中的第一

推荐阅读

- 三面字节跳动被虐得“体无完肤”，15天读完这份pdf，终拿下美团研发
阅读 79,827
- 对于二本渣渣来说，面试阿里P6也太难了！（两年crud经验，已拿
阅读 45,313
- 面试官再问你 HashMap 底层原理，就把这篇文章甩给他看
阅读 5,157
- 离开菜鸟&新的面试体验
阅读 9,364
- 败给“MySQL”的第33天，我重振旗鼓，四面拿下阿里淘系offer
阅读 6,312

条指令中，`jmp *0x8049684`（跳转到GOT[4]内容为：804846a）又回到了pushl指令处将addvec压入栈中。然后通过`jmp 8048444`跳转到PLT[0]动态链接器中。动态链接器通过两个栈条目来确定addvec的位置，用这个地址覆盖GOT[4]，并把控制权转到addvec。

下次访问的方式还是通过传递控制权到PLT[2]中，但这次得到的GOT[4]的地址已经被延迟绑定好了。这样唯一的开销就是间接跳转。



17人点赞 >

《深入理解计算机系统》

...

"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下



唐鱼的学习探索 如果我像一般人一样读那么多书，我就跟他们一样愚蠢了。
总资产29 (约2.85元) 共写了10.4W字 获得530个赞 共463个粉丝

关注



写下你的评论...

评论1 赞17 ...



写下你的评论...

全部评论 1 只看作者 按时间倒序 按时间正序



周周末
2楼 2017.04.01 20:12

666

赞 回复

被以下专题收入，发现更多相似内容

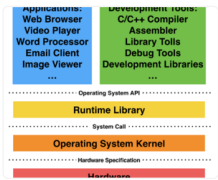
- 《深入理解计算...
- 程序员
- 操作系统
- 操作系统原理

推荐阅读 更多精彩内容 >

iOS 程序员的自我修养 — 读《程序员的自我修养-链接、装载...

2016年国庆假期终于把此书过完，整理笔记和体会于此。关于书名 书名源于俄罗斯的演员斯坦尼斯拉夫斯基创作的《演员...

李剑飞的简书 阅读 4,247 评论 1 赞 53



读书笔记 - 《程序员的自我修养》

一、温故而知新 1. 内存不够怎么办 内存简单分配策略的问题地址空间不隔离内存使用效率低程序运行的地址不确定 关于...

SeanCST 阅读 3,726 评论 0 赞 21

Android - 收藏集

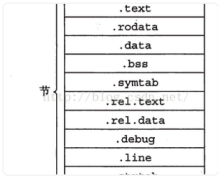
Android 自定义View的各种姿势1 Activity的显示之ViewRootImpl详解 Activity...

passiontim 阅读 135,746 评论 17 赞 578

编译链接

转自<http://blog.csdn.net/navyhu/article/details/47023317>理解链...

扎Zn了老Fe 阅读 404 评论 0 赞 0



还来不及爱，就走远，哪怕在给我多一点的时间，去爱！

每一次离别，都是一种痛苦的蔓延，我承受不住太多的牵绊，去年的十一月，我家的马林那，带着他的宝贝，离开了我们，家里的...

I安 阅读 46 评论 0 赞 0