讀《深入理解計算機系統》第七章《鏈接》筆記。

在ELF文件的動態連接機制中，每一個外部定義的符號在全局偏移表 (Global Offset Table，GOT)中有相應的條目，如果符號是函數則在過程連接表(Procedure Linkage Table，PLT)中也有相應的條目，且一個PLT條目對應一個GOT條目。以簡單的例子觀察GOT與PLT的變化機制。下面是一個簡單的Hello World程序：

```
#include <stdio.h>

void showmsg(char *szMsg
                {
    printf("%s\n", szMsg);
                }

int main(int argc, char **argv)
                {
    char szMsg[] = "Hello,
                world!";
        showmsg(szMsg);

                return 0;
                }
```

使用gcc編譯：

```
gcc -o a.out -g
helloworld.c
```

使用objdump -S a.out查看反彙編代碼：

```
void showmsg(char *szMsg)
{
 8048434:    55              push   %ebp
 8048435:    89 e5           mov    %esp,%ebp
 8048437:    83 ec 18        sub    $0x18,%esp
    printf("%s\n", szMsg);
 804843a:    8b 45 08        mov
0x8(%ebp),%eax
 804843d:    89 04 24        mov    %eax,(%esp)
 8048440:    e8 0b ff ff ff  call   8048350
<puts@plt>
}
 8048445:    c9              leave
 8048446:    c3              ret
```

對printf的調用被編譯器改成了puts@plt，位於0×08048350，這是一個PLT（Procedure Linkage Table）條目，往上翻查看這個地址的代碼：

```
08048350 <puts@plt>:
 8048350:       ff 25 04 a0 04 08       jmp    *0x804a004
 8048356:       68 08 00 00 00          push   $0x8
 804835b:       e9 d0 ff ff ff          jmp    8048330
<_init+0x38>
```

第一條指令跳轉到0x0804a004地址處的值去執行，實際上0x0804a004就是一個對應的GOT（Global Offset Table）條目的位置了。使用

```
objdump -R
a.out
```

可以看到：

```
a.out:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   __stack_chk_fa
0804a004 R_386_JUMP_SLOT   puts
0804a008 R_386_JUMP_SLOT   __gmon_start_
0804a00c R_386_JUMP_SLOT
__libc_start_main
```

就是puts那一條的offset的值。

下面用gdb動態調試a.out，查看GOT和PLT條目的變化：

```
gdb a.out
(gdb) l
1       #include <stdio.h>
2
3       void showmsg(char *szMsg)
4       {
5               printf("%s\n", szMsg);
6       }
7
8       int main(int argc, char **argv)
9       {
10              char szMsg[] = "Hello, world!";
(gdb) b main
```

```
Breakpoint 1 at 0x8048457: file helloworld.c, line 9.
(gdb) r
Starting program: /home/winson/linuxc/a.out

Breakpoint 1, main (argc=1, argv=0xbffff394) at
helloworld.c:9
9       {
```

現在看一下地址0x804a004里的值是多少：

```
(gdb) x 0x804a004
0x804a004 <puts@got.plt>:   0x08048356
```

就是puts@plt的第二條指令了，可以查看puts@plt的代碼，或者反彙編0×08048356處的代碼：

```
(gdb) disas 0x08048356
Dump of assembler code for function
puts@plt:
   0x08048350 <+0>:  jmp    *0x804a004
   0x08048356 <+6>:  push   $0x8
   0x0804835b <+11>:jmp    0x8048330
End of assembler dump.
```

其實第一條指令執行後就是jmp到了第二條指令。接著push一個標識，這裡是0×08，**然後再次jmp來確定真正的函數地址**。第一次執行完以後，會修改GOT條目的內容（這裡就是修改0x0804a004的內容了）。

給showmsg下斷點，讓其執行完printf之後，再次查看GOT條目的內容：

```
(gdb) b 5
Breakpoint 2 at 0x804843a: file helloworld.c, line 5.
(gdb) c
Continuing.

Breakpoint 2, showmsg (szMsg=0xbffff2de "Hello, world!") at
helloworld.c:5
5               printf("%s\n", szMsg);
(gdb) n
Hello, world!
6       }
(gdb) x 0x0804a004
0x804a004 <puts@got.plt>:   0x001913b0
```

可以看到現在GOT條目直接指向真正的函數了，也就是之後的調用，先跳轉到PLT條目，然後經由GOT條目直接跳轉到函數代碼，比之前少了一些指令，更加快速。

其實就是第一次調用函數之後，會修改GOT條目的內容，第一次相對來說慢一點，以後快一點。有點像Windows下PE文件的導出表條目中的FirstThunk和OriginalFirstThunk的東西。Linux的東西剛接觸，如有錯誤，歡迎指正。

2014-05-07 更正

舊文章誤人子弟了。PLT其實是延遲綁定技術，也就是等到調用函數的時候才進行函數地址的定位。示例代碼：

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804841d <+0>:  push   %ebp
   0x0804841e <+1>:  mov    %esp,%ebp
   0x08048420 <+3>:  and    $0xfffffff0,%esp
   0x08048423 <+6>:  sub    $0x10,%esp
   0x08048426 <+9>:  movl   $0x80484d0,(%esp)
   0x0804842d <+16>:call   0x80482f0 <puts@plt>  ; 跟蹤0x080482f0
   0x08048432 <+21>:mov    $0x0,%eax
   0x08048437 <+26>:leave
   0x08048438 <+27>:ret
End of assembler dump.

(gdb) b *0x0804842d
Breakpoint 1 at 0x804842d
(gdb) r
Starting program: /home/winson/Documents/test

Breakpoint 1, 0x0804842d in main ()
(gdb) disas main
Dump of assembler code for function main:
   0x0804841d <+0>:  push   %ebp
   0x0804841e <+1>:  mov    %esp,%ebp
   0x08048420 <+3>:  and    $0xfffffff0,%esp
   0x08048423 <+6>:  sub    $0x10,%esp
   0x08048426 <+9>:  movl   $0x80484d0,(%esp)
=> 0x0804842d <+16>:        call   0x80482f0 <puts@plt>  ; 跟蹤0x080482f0
   0x08048432 <+21>:mov    $0x0,%eax
   0x08048437 <+26>:leave
   0x08048438 <+27>:ret
End of assembler dump.
```

```
(gdb) disas 0x080482f0
Dump of assembler code for function puts@plt:
   0x080482f0 <+0>:   jmp    *0x804a00c        ; 查看0x804a00c內存
   0x080482f6 <+6>:   push   $0x0
   0x080482fb <+11>: jmp    0x80482e0
End of assembler dump.

(gdb) x 0x0804a00c
0x804a00c <puts@got.plt>:    0x080482f6        ; 實際上是跳轉回去執行下一條指令

(gdb) x /30i 0x080482e0
   0x80482e0:  pushl  0x804a004
   0x80482e6:  jmp    *0x804a008              ; 跟蹤進入
   0x80482ec:  add    %al,(%eax)
   0x80482ee:  add    %al,(%eax)
   0x80482f0 <puts@plt>:      jmp    *0x804a00c
   0x80482f6 <puts@plt+6>:    push   $0x0
   0x80482fb <puts@plt+11>:   jmp    0x80482e0
   ...

(gdb) x /1xw 0x0804a008
0x804a008:    0xb7ff24f0

(gdb) b *0x080482e6
Breakpoint 2 at 0x80482e6
(gdb) c
Continuing.

Breakpoint 2, 0x080482e6 in ?? ()
(gdb) ni
_dl_runtime_resolve () at ../sysdeps/i386/dl-trampoline.S:28
28      ../sysdeps/i386/dl-trampoline.S: No such file or directory.
(gdb) i r eip
eip            0xb7ff24f0  0xb7ff24f0 <_dl_runtime_resolve>
(gdb)
```

可以看到第一次其實是調用_dl_runtime_resolve進行GOT對應的表項的修改，以後就直接從GOT調轉到真正的函數了。更詳細的內容可以參考《程序員的自我修養》第200頁**延遲綁定（PLT）**。