Position Independent Code (PIC) in shared libraries on x64 (https://eli.thegreenplace.net/2011/11/11/position-independent-code-pic-in-shared-libraries-on-x64)

November 11, 2011 at 15:10 **Tags** Articles (https://eli.thegreenplace.net/tag/articles) , Assembly (https://eli.thegreenplace.net/tag/assembly) , C & C++ (https://eli.thegreenplace.net/tag/c-c) , Linkers and loaders (https://eli.thegreenplace.net/tag/linkers-and-loaders) , Linux (https://eli.thegreenplace.net/tag/linux)

The previous article (http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/) explained how position independent code (PIC) works, with code compiled for the x86 architecture as an example. I promised to cover PIC on x64 [1] in a separate article, so here we are. This article will go into much less detail, since it assumes an understanding of how PIC works in theory. In general, the idea is similar for both platforms, but some details differ because of unique features of each architecture.

RIP-relative addressing

On x86, while function references (with the `call` instruction) use relative offsets from the instruction pointer, data references (with the `mov` instruction) only support absolute addresses. As we've seen in the previous article, this makes PIC code somewhat less efficient, since PIC by its nature requires making all offsets IP-relative; absolute addresses and position independence don't go well together.

x64 fixes that, with a new "RIP-relative addressing mode", which is the default for all 64-bit `mov` instructions that reference memory (it's used for other instructions as well, such as `lea`). A quote from the "Intel Architecture Manual vol 2a":

> A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

The displacement used in RIP-relative mode is 32 bits in size. Since it should be useful for both positive and negative offsets, roughly +/- 2GB is the maximal offset from RIP supported by this addressing mode.

x64 PIC with data references - an example

For easier comparison, I will use the same C source as in the data reference example of the previous article:

```
int myglob = 42;

int ml_func(int a, int b)
{
    return myglob + a + b;
}
```

Let's look at the disassembly of `ml_func`:

```
0000000000005ec <ml_func>:
 5ec:   55                      push   rbp
 5ed:   48 89 e5                mov    rbp,rsp
 5f0:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
 5f3:   89 75 f8                mov    DWORD PTR [rbp-0x8],esi
 5f6:   48 8b 05 db 09 20 00    mov    rax,QWORD PTR [rip+0x2009db]
 5fd:   8b 00                   mov    eax,DWORD PTR [rax]
 5ff:   03 45 fc                add    eax,DWORD PTR [rbp-0x4]
 602:   03 45 f8                add    eax,DWORD PTR [rbp-0x8]
 605:   c9                      leave
 606:   c3                      ret
```

The most interesting instruction here is at `0x5f6`: it places the address of `myglobal` into `rax`, by referencing an entry in the GOT. As we can see, it uses RIP relative addressing. Since it's relative to the address of the next instruction, what we actually get is `0x5fd + 0x2009db = 0x200fd8`. So the GOT entry holding the address of `myglob` is at `0x200fd8`. Let's check if it makes sense:

```
$ readelf -S libmlpic_dataonly.so
There are 35 section headers, starting at offset 0x13a8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align

[...]
  [20] .got              PROGBITS         0000000000200fc8  00000fc8
       0000000000000020  0000000000000008  WA       0     0     8
[...]
```

GOT starts at `0x200fc8`, so `myglob` is in its third entry. We can also see the relocation inserted for the GOT reference to `myglob`:

```
$ readelf -r libmlpic_dataonly.so

Relocation section '.rela.dyn' at offset 0x450 contains 5 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
[...]
000000200fd8  000500000006 R_X86_64_GLOB_DAT 0000000000201010 myglob + 0
[...]
```

Indeed, a relocation entry for `0x200fd8` telling the dynamic linker to place the address of `myglob` into it once the final address of this symbol is known.

So it should be quite clear how the address of `myglob` is obtained in the code. The next instruction in the disassembly (at `0x5fd`) then dereferences the address to get the value of `myglob` into `eax` [2].

x64 PIC with function calls - an example

Now let's see how function calls work with PIC code on x64. Once again, we'll use the same example from the previous article:

```
int myglob = 42;

int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}
```

Disassembling `ml_func`, we get:

```
000000000000064b <ml_func>:
 64b:   55                      push   rbp
 64c:   48 89 e5                mov    rbp,rsp
 64f:   48 83 ec 20             sub    rsp,0x20
 653:   89 7d ec                mov    DWORD PTR [rbp-0x14],edi
 656:   89 75 e8                mov    DWORD PTR [rbp-0x18],esi
 659:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
 65c:   89 c7                   mov    edi,eax
 65e:   e8 fd fe ff ff          call   560 <ml_util_func@plt>
 [... snip more code ...]
```

The call is, as before, to `ml_util_func@plt`. Let's see what's there:

```
0000000000000560 <ml_util_func@plt>:
 560:   ff 25 a2 0a 20 00       jmp    QWORD PTR [rip+0x200aa2]
 566:   68 01 00 00 00          push   0x1
 56b:   e9 d0 ff ff ff          jmp    540 <_init+0x18>
```

So, the GOT entry holding the actual address of `ml_util_func` is at `0x200aa2 + 0x566 = 0x201008`.

And there's a relocation for it, as expected:

```
$ readelf -r libmlpic.so

Relocation section '.rela.dyn' at offset 0x480 contains 5 entries:
[...]

Relocation section '.rela.plt' at offset 0x4f8 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
[...]
000000201008  000600000007 R_X86_64_JUMP_SLO 000000000000063c ml_util_func + 0
```

Performance implications

In both examples, it can be seen that PIC on x64 requires less instructions than on x86. On x86, the GOT address is loaded into some base register (`ebx` by convention) in two steps - first the address of the instruction is obtained with a special function call, and then the offset to GOT is added. Both steps aren't required on x64, since the relative offset to GOT is known to the linker and can simply be encoded in the instruction itself with RIP relative addressing.

When calling a function, there's also no need to prepare the GOT address in `ebx` for the trampoline, as the x86 code does, since the trampoline just accesses its GOT entry directly through RIP-relative addressing.

So PIC on x64 still requires extra instructions when compared to non-PIC code, but the additional cost is smaller. The indirect cost of tying down a register to use as the GOT pointer (which is painful on x86) is also gone, since no such register is needed with RIP-relative addressing [3]. All in all, x64 PIC results in a much smaller performance hit than on x86, making it much more attractive. So attractive, in fact, that it's the default method for writing shared libraries for this architecture.

Extra credit: Non-PIC code on x64

Not only does `gcc` encourage you to use PIC for shared libraries on x64, it requires it by default. For instance, if we compile the first example without `-fpic` [4] and then try to link it into a shared library (with `-shared`), we'll get an error from the linker, something like this:

```
/usr/bin/ld: ml_nopic_dataonly.o: relocation R_X86_64_PC32 against symbol `myglob' can not be
/usr/bin/ld: final link failed: Bad value
collect2: ld returned 1 exit status
```

What's going on? Let's look at the disassembly of `ml_nopic_dataonly.o` [5]:

```
0000000000000000 <ml_func>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
   7:   89 75 f8                mov    DWORD PTR [rbp-0x8],esi
   a:   8b 05 00 00 00 00       mov    eax,DWORD PTR [rip+0x0]
  10:   03 45 fc                add    eax,DWORD PTR [rbp-0x4]
  13:   03 45 f8                add    eax,DWORD PTR [rbp-0x8]
  16:   c9                      leave
  17:   c3                      ret
```

Note how `myglob` is accessed here, in instruction at address `0xa`. It expects the linker to patch in a relocation to the actual location of `myglob` into the operand of the instruction (so no GOT redirection is required):

```
$ readelf -r ml_nopic_dataonly.o

Relocation section '.rela.text' at offset 0xb38 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000c  000f00000002 R_X86_64_PC32     0000000000000000 myglob - 4
[...]
```

Here is the `R_X86_64_PC32` relocation the linker was complaining about. It just can't link an object with such relocation into a shared library. Why? Because the displacement of the `mov` (the part that's added to `rip`) must fit in 32 bits, and when a code gets into a shared library, we just can't know in advance that 32 bits will be enough. After all, this is a full 64-bit architecture, with a vast address space. The symbol may eventually be found in some shared library that's farther away from the reference than 32 bits will allow to reference. This makes `R_X86_64_PC32` an invalid relocation for shared libraries on x64.

But can we still somehow create non-PIC code on x64? Yes! We should be instructing the compiler to use the "large code model", by adding the `-mcmodel=large` flag. The topic of code models is interesting, but explaining it would just take us too far from the real goal of this article [6]. So I'll just say briefly that a code model is a kind of agreement between the programmer and the compiler, where the programmer makes a certain promise to the compiler about the size of offsets the program will be using. In exchange, the compiler can generate better code.

It turns out that to make the compiler generate non-PIC code on x64 that actually pleases the linker, only the large code model is suitable, because it's the least restrictive. Remember how I explained why the simple relocation isn't good enough on x64, for fear of an offset which will get farther than 32 bits away during linking? Well, the large code model basically gives up on all offset assumptions and uses the

largest 64-bit offsets for all its data references. This makes load-time relocations always safe, and enables non-PIC code generation on x64. Let's see the disassembly of the first example compiled without `-fpic` and with `-mcmodel=large`:

```
0000000000000000 <ml_func>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
   7:   89 75 f8                mov    DWORD PTR [rbp-0x8],esi
   a:   48 b8 00 00 00 00 00    mov    rax,0x0
  11:   00 00 00
  14:   8b 00                   mov    eax,DWORD PTR [rax]
  16:   03 45 fc                add    eax,DWORD PTR [rbp-0x4]
  19:   03 45 f8                add    eax,DWORD PTR [rbp-0x8]
  1c:   c9                      leave
  1d:   c3                      ret
```

The instruction at address `0xa` places the address of `myglob` into `rax`. Note that its argument is currently `0`, which tells us to expect a relocation. Note also that it has a full 64-bit address argument. Moreover, the argument is absolute and not RIP-relative [7]. Note also that two instructions are actually required here to get the *value* of `myglob` into `eax`. This is one reason why the large code model is less efficient than the alternatives.

Now let's see the relocations:

```
$ readelf -r ml_nopic_dataonly.o

Relocation section '.rela.text' at offset 0xb40 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000c  000f00000001 R_X86_64_64       0000000000000000 myglob + 0
[...]
```

Note the relocation type has changed to `R_X86_64_64`, which is an absolute relocation that can have a 64-bit value. It's acceptable by the linker, which will now gladly agree to link this object file into a shared library.

Some judgmental thinking may bring you to ponder why the compiler generated code that isn't suitable for load-time relocation by default. The answer to this is simple. Don't forget that code also tends to get directly linked into executables, which don't require load-time relocations at all. Therefore, by default the compiler assumes the small code model to generate the most efficient code. If you know your code is going to get into a shared library, and you don't want PIC, then just tell it to use the large code model explicitly. I think `gcc`'s behavior makes sense here.

Another thing to think about is why there are no problems with PIC code using the small code model. The reason is that the GOT is always located in the same shared library as the code that references it, and unless a single shared library is big enough for a 32-bit address space, there should be no problems addressing the PIC with 32-bit RIP-relative offsets. Such huge shared libraries are unlikely, but in case you're working on one, the AMD64 ABI has a "large PIC code model" for this purpose.

Conclusion

This article complements its predecessor by showing how PIC works on the x64 architecture. This architecture has a new addressing mode that helps PIC code be faster, and thus makes it more desirable for shared libraries than on x86, where the cost is higher. Since x64 is currently the most popular architecture used in servers, desktops and laptops, this is important to know. Therefore, I tried to focus on additional aspects of compiling code into shared libraries, such as non-PIC code. If you have any questions and/or suggestions on future directions to explore, please let me know in the comments or by email.

[1] As always, I'm using x64 as a convenient short name for the architecture known as x86-64, AMD64 or Intel 64.

[2] Into `eax` and not `rax` because the type of `myglob` is `int`, which is still 32-bit on x64.

[3] By the way, it would be much less "painful" to tie down a register on x64, since it has twice as many GPRs as x86.

[4] It also happens if we explicitly specify we don't want PIC by passing `-fno-pic` to `gcc`.

[5] Note that unlike other disassembly listings we've been looking at in this and the previous article, this is an object file, not a shared library or executable. Therefore it will contain some relocations for the linker.

[6] For some good information on this subject, take a look at the AMD64 ABI, and `man gcc`.

[7] Some assemblers call this instruction `movabs` to distinguish it from the other `mov` instructions that accept a relative argument. The Intel architecture manual, however, keeps naming it just `mov`. Its opcode format is `REX.W + B8 + rd`.

For comments, please send me ✉ an email (mailto:eliben@gmail.com).