x86-64 Code Model

This article describes differences in the OS X x86-64 user-space code model from the code model described in *System V Application Binary Interface AMD64 Architecture Processor Supplement*, at http://www.x86-64.org/documentation.

The x86-64 environment in OS X has only one code model for user-space code. It's most similar to the small PIC model defined by the x86-64 System V ABI. Under Mach-O, all static initialized storage (both code and data) must fit within a 4GB Mach-O file. Uninitialized (zero-fill) data may be any size, although there is a practical limit imposed by OS X.

All local and small data is accessed directly using addressing that's relative to the instruction pointer (RIP-relative addressing). All large or possibly nonlocal data is accessed indirectly through a global offset table (GOT) entry. The GOT entry is accessed directly using RIP-relative addressing.

Listing 1 shows sample C code and corresponding assemble code.

Listing 1 C code and the corresponding assembly code

```
extern int src[];
extern int dst[];
extern int* ptr;

static int lsrc[500];
static int ldst[500];
static int bsrc[500000];
static int bdst[500000];
static int* lptr;

dst[0] = src[0];        movq _src@GOTPCREL(%rip), %rax
                        movl (%rax), %edx
                        movq _dst@GOTPCREL(%rip), %rax
                        movl %edx, (%rax)

ptr = dst;              movq _dst@GOTPCREL(%rip), %rdx
                        movq _ptr@GOTPCREL(%rip), %rax
                        movq %rdx, (%rax)
                        ret

*ptr = src[0];          movq  _ptr@GOTPCREL(%rip), %rax
                        movq   (%rax), %rdx
                        movq   _src@GOTPCREL(%rip), %rax
                        movl   (%rax), %eax
                        movl   %eax, (%rdx)
                        ret
```

```
ldst[0] = lsrc[0];      movl _lsrc(%rip), %eax
                        movl %eax, _ldst(%rip)

lptr = ldst;            lea  _ldst(%rip), %rax
                        movq %rax, _lptr(%rip)

*lptr = lsrc[0];        movl _lsrc(%rip), %edx
                        movq _lptr(%rip), %rax
                        movl %edx, (%rax)

bdst[0] = bsrc[0];      movq _bsrc@GOTPCREL(%rip), %rax
                        movl (%rax), %edx
                        movq _bdst@GOTPCREL(%rip), %rax
                        movl %edx, (%rax)

lptr = bdst;            movq _bdst@GOTPCREL(%rip), %rax
                        movq %rax, _lptr(%rip)

*lptr = bsrc[0];        movq _bsrc@GOTPCREL(%rip), %rdx
                        movq _lptr(%rip), %rax
                        movl (%rdx), %edx
```

The OS X x86-64 code-generation model accesses large local data through the GOT, which is different from the way the small PIC model works in the System V x86-64 environment. Indirection through the GOT obviates the need for a medium code model. This behavior stems from the fact that, when the linker lays out data, it places data that is accessed directly (small local data and the GOT itself) within 2 GB of the code. Other data can be placed farther away because these data are accessed only through the GOT. This behavior enables large (greater than 4 GB) and small executables to be built using the same code model.

Note: It is acceptable for the compiler to access static data through a GOT entry. The linker preserves the static semantics of the symbol.

The code model for function calls is very simple, as shown in Listing 2.

Listing 2 The code model for function calls

```
extern int foo();
static int bar();

foo();                    call _foo

bar();                    call _bar
```

All direct function calls are made using the `CALL rel32` instruction.

The linker is responsible for creating GOT entries (also known as nonlazy pointers) as well as stub functions and lazy pointers (also known as program load table entries, or PLT entries) for calls to another linkage unit. Since the linker must create these entries, it can also choose not to create them when it sees the opportunity. The linker has a complicated set of rules that dictate which symbols must be accessed indirectly (depending on flat versus two-level namespace, weak versus non-weak definitions, symbol visibility, distance from code, and so on). But ultimately there are many symbols that can be accessed directly (not through GOT or PLT entries). For these symbols the linker makes the following optimization:

1. A `CALL` or `JMP` instruction performs a direct, PC-relative branch to the target.
2. A load instruction performed on a GOT entry (for example, `movq _foo@GOTPCREL(%rip), %rxx`) is transformed into a `LEA` calculation (for example, `leaq _foo(%rip), %rxx`). This transformation removes one GOT entry and saves one memory load.

In both cases special relocations are used that allow the linker to perform this optimization.