Indirect Addressing

Indirect addressing is the name of the code generation technique that allows symbols defined in one file to be referenced from another file, without requiring the referencing file to have explicit knowledge of the layout of the file that defines the symbol. Therefore, the defining file can be modified independently of the referencing file. Indirect addressing minimizes the number of locations that must be modified by the dynamic linker, which facilitates code sharing and improves performance.

When a file uses data that is defined in another file, it creates symbol references. A *symbol reference* identifies the file from which a symbol is imported and the referenced symbol. There are two types of symbol references: non lazy and lazy.

- *Non-lazy symbol references* are resolved (bound to their definitions) by the dynamic linker when a module is loaded.

  A non-lazy symbol reference is essentially a *symbol pointer*—a pointer-sized piece of data. The compiler generates non-lazy symbol references for data symbols or function addresses.

- *Lazy symbol references* are resolved by the dynamic linker the first time they are used (not at load time). Subsequent calls to the referenced symbol jump directly to the symbol's definition.

  Lazy symbol references are made up of a symbol pointer and a *symbol stub*, a small amount of code that directly dereferences and jumps through the symbol pointer. The compiler generates lazy symbol references when it encounters a call to a function defined in another file.

The following sections describe how symbol references are implemented for the PowerPC and IA-32 architectures. For detailed information on the PowerPC and IA-32 symbol stubs, see *OS X Assembler Reference*.

PowerPC Symbol References

In the PowerPC architecture, when generating calls to functions that are defined in other files, the compiler creates a symbol stub and a lazy symbol pointer. The *lazy symbol pointer* is an address that is initially set to glue code that calls the linker glue function `dyld_stub_binding_helper`. This glue function calls the dynamic linker function that performs the actual work of binding the stub. On return from `dyld_stub_binding_helper`, the lazy pointer points to the actual address of the external function.

The simple code example in Listing 1 might produce two different types of symbol stubs, depending on whether it is compiled with position-independent code generation. Listing 2 shows indirect addressing without position-independent code, and Listing 3 shows both indirect addressing and position-independent code.

Listing 1 C code example for indirect function calls

```
extern void bar(void);
void foo(void)
{
    bar();
}
```

Listing 2 Example of an indirect function call

```
.text
        ; The function foo
        .align 2
        .globl _foo
_foo:
        mflr r0         ; move the link register into r0
        stw r0,8(r1)    ; save the link register value on the stack
        stwu r1,-64(r1) ; set up the frame on the stack
        bl L_bar$stub   ; branch and link to the symbol stub for _bar
        lwz r0,72(r1)   ; load the link register value from the stack
        addi r1,r1,64   ; removed the frame from the stack
        mtlr r0         ; restore the link register
        blr             ; branch to the link register to return

.symbol_stub            ; the standard symbol stub section
L_bar$stub:
        .indirect_symbol _bar               ; identify this symbol stub for the
                                            ;  symbol _bar
        lis r11,ha16(L_bar$lazy_ptr)        ; load r11 with the high 16 bits of the
                                            ;  address of bar’s lazy pointer
        lwz r12,lo16(L_bar$lazy_ptr)(r11)   ; load the value of bar’s lazy pointer
                                            ;  into r12
        mtctr r12                           ; move r12 to the count register
        addi r11,r11,lo16(L_bar$lazy_ptr)   ; load r11 with the address of bars lazy
                                            ; pointer
        bctr                                ; jump to the value in bar’s lazy pointer

.lazy_symbol_pointer    ; the lazy pointer section
L_bar$lazy_ptr:
        .indirect_symbol _bar               ; identify this lazy pointer for symbol
                                            ;  _bar
        .long dyld_stub_binding_helper      ; initialize the lazy pointer to the stub
                                            ;  binding helper address
```

Listing 3 Example of a position-independent, indirect function call

```
.text
        ; The function foo
        .align 2
        .globl _foo
_foo:
        mflr r0         ; move the link register into r0
        stw r0,8(r1)    ; save the link register value on the stack
        stwu r1,-80(r1) ; set up the frame on the stack
        bl L_bar$stub   ; branch and link to the symbol stub for _bar
        lwz r0,88(r1)   ; load the link register value from the stack
        addi r1,r1,80   ; removed the frame from the stack
        mtlr r0         ; restore the link register
        blr             ; branch to the link register to return

.picsymbol_stub         ; the standard pic symbol stub section
L_bar$stub:
        .indirect_symbol _bar       ; identify this symbol stub for the symbol _bar
        mflr r0                     ; save the link register (LR)
        bcl 20,31,L0$_bar           ; Use the branch-always instruction that does not
                                    ;  affect the link register stack to get the
                                    ;  address of L0$_bar into the LR.
L0$_bar:
        mflr r11                                ; then move LR to r11
                                                ; bar’s lazy pointer is located at
                                                ;  L0$_bar + distance
        addis r11,r11,ha16(L_bar$lazy_ptr-L0$_bar); L0$_bar plus high 16 bits of
                                                ;  distance
        mtlr r0                                 ; restore the previous LR
        lwz r12,lo16(L_bar$lazy_ptr-L0$_bar)(r11); ...plus low 16 of distance
        mtctr r12                               ; move r12 to the count register
        addi r11,r11,lo16(L_bar$lazy_ptr-L0$_bar); load r11 with the address of bar’s
                                                ; lazy pointer
        bctr                                    ; jump to the value in bar’s lazy
                                                ;  pointer

.lazy_symbol_pointer    ; the lazy pointer section
L_bar$lazy_ptr:
        .indirect_symbol _bar           ; identify this lazy pointer for symbol bar
        .long dyld_stub_binding_helper  ; initialize the lazy pointer to the stub
                                        ;  binding helper address.
```

As you can see, the `__picsymbol_stub` code in Listing 3 resembles the position-independent code generated for Listing 2. For any position-independent Mach-O file, symbol stubs must obviously be position independent, too.

The static linker performs two optimizations when writing output files:

- It removes symbol stubs for references to symbols that are defined in the same module, modifying branch instructions that were calling through stubs to branch directly to the call.
- It removes duplicates of the same symbol stub, updating branch instructions as necessary.

Note that a routine that branches indirectly to another routine must store the target of the call in GPR11 or GPR12. Standardizing the registers used by the compiler to store the target address makes it possible to optimize dynamic code generation. Because the target address needs to be stored in a register in any event, this convention standardizes what register to use. Routines that may have been called directly should not depend on the value of GR12 because, in the case of a direct call, its value is not defined.

IA-32 Symbol References

In the IA-32 architecture, symbol references are implemented as a symbol stub and a lazy symbol pointer combined into one `JMP` instruction. Initially, such instructions point to the dynamic linker. When the dynamic linker encounters such an instruction, it locates the referenced symbol and modifies the `JMP` instruction to point directly to this symbol. Therefore, subsequent executions of the `JMP` instruction jump directly to the referenced symbol.

Listing 4 and Listing 5 show a simple C program and the IA-32 assembly generated highlighting the symbol stub and non-lazy pointer for an imported symbol.

Listing 4 C program using an imported symbol

```
#include <stdio.h>
main( int arc, char *argv[])
{
  fprintf(stdout, "hello, world!\n") ;
}
```

Listing 5 IA-32 symbol reference in assembly

```
        .cstring
LC0:
        .ascii "hello, world!\12\0"
        .text
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    L___sF$non_lazy_ptr, %eax
        addl    $88, %eax
        movl    %eax, 12(%esp)
        movl    $14, 8(%esp)
        movl    $1, 4(%esp)
        movl    $LC0, (%esp)
        call    L_fwrite$stub                         ; call to imported symbol
        leave
        ret
        .section
__IMPORT,__jump_table,symbol_stubs,self_modifying_code+pure_instructions,5
L_fwrite$stub:                                        ; symbol stub
        .indirect_symbol _fwrite
        hlt ; hlt ; hlt ; hlt ; hlt
        .section __IMPORT,__pointers,non_lazy_symbol_pointers
```

```
L___sF$non_lazy_ptr:                                 ; nonlazy pointer
        .indirect_symbol ___sF
        .long   0
        .subsections_via_symbols
```

x86-64 Symbol References

This section describes deviations from the System V x85-64 environment in the area of symbol references.

Note: The OS X x86-64 environment uses Mach-O (not ELF) as its executable file format.

The static linker is responsible for generating all stub functions, stub helper functions, lazy and non-lazy pointers, as well as the indirect symbol table needed by the dynamic loader (`dyld`).

For reference, Listing 6 shows how a a stub, helper, and lazy pointer are generated.

Listing 6 Generating a stub, helper and lazy pointer

```
_foo$stub:             jmp     *_foo$lazy_pointer(%rip)
_foo$stub_helper:      leaq     _foo$lazy_pointer(%rip),%r11
                       jmp      dyld_stub_binding_helper
_foo$lazy_pointer:     .quad    _foo$stub_helper
```