Link 9. Position Independent Code

Young W. Lim

2018-12-04 Tue

Outline

1. Linking - 9. Position Independent Code
   - Based on
   - Position Independent Code
   - PIC Data References
   - PIC Function Calls

Based on

"Self-service Linux: Mastering the Art of Problem Determination",
Mark Wilding
"Computer Architecture: A Programmer's Perspective",
Bryant & O'Hallaron

Position Independent Code

1. Sharing the same library code in memory

Sharing the same library code in memory

- library code can be loaded and executed
  at any address without modification by the linker
- no *a priori* dedicated portion of the address space
- `-fPIC` in gcc

Sharing codes on IA32 system

- calls to procedures in the same object
  - no relocation
  - PC-relaive with know offsets
  - already PIC

- calls to externally defined procedures
  references to global variables
  - need relocation at link time
  - normally not PIC

PIC Data References

1. Accessing global variables
2. Global Offset Table (GOT)
3. Indirect reference through the GOT
4. Global variable access using the GOT

Accessing global variables (1)

- Compilers generates PIC references to global variables utilizing the follwing fact
- No matter where an object module is loaded in memory including the shared object modules,
  the data segment is always allocated *immediately after* the code segment
-
  ```
  +--------------------+------------------+-----------------------+
  | Read/Write segment | higher addresses | .data, .bss           |
  +--------------------+------------------+-----------------------+
  | Read-only segment  | lower addresses  | .init, .text, .rodata |
  +--------------------+------------------+-----------------------+
  starting from 0x08048000
  ```

Accessing global variables (2)

- the distance between
  - any instruction in the code segment and
  - any variable inthe data segment

  is a run-time constant
  - independent of the absolute memory locations of code and data segments
- Global Offset Table (GOT) at the beginning of data segment

Global Offset Table (GOT)

- GOT contains an entry
  - for each global data object
    that is referenced by the object module
- the compiler generates also a relocation record
  - for each entry in the GOT
- at load time, the dynamic linker relocates
  each entry in the GOT
  so that it contains the appropriate absolute address
- *each* object module that references global data has *its own* GOT

Indirect reference through the GOT

- at run time, each global variable is referenced *indirectly* through the GOT
- PIC code incurs *performance degradation*
  - each global variable reference require 5 instructions
  - additional memory reference to the GOT
  - machines with large register files can overcome this disadvantages
  - on register demanding IA32 systems, losing even one register can cause to spill the registers to the stack
- a pattern of codes

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

Global variable access using the GOT (1)

- the call to LL pushes the return address
  the address of `popl` instruction on the stack
- then `popl` instruction pops this address into %ebx
- the result of these 2 instructions
  to move the value of the PC into register %ebx

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

Global variable access using the GOT (2)

- addl adds a constant offest to %ebx so that it points to the appropriate entry in the GOT where the absolute address can be fetched
- now, the global variable can be accessed indirectly through the GOT entry contained in %ebx
- the 2 movl load the contents of the global variable indirectly through the GOD into register %eax

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

PIC Function Calls

1. Resolving external procedure calls
2. Lazy Binding
3. Vector addition and multiplication examples
4. The Global Offset Table for the previous examples
5. The Global Offset Table Example
6. Procedure Linkage Table
7. Procedure Linkage Table for the previous examples
8. GOT and PLT for addvec
9. Procedure Linkage Table Example

Resolving external procedure calls

- the same approach to the PIC references to global variables
  - this approach require 3 additional instructions
- ELF compilation systems use *lazy binding* technique
  - defers the binding of procedure addresses
    until the first time the procedure is actually called
  - significant run-time overhead the first time call
  - for subsequent calls
    - one additional instruction
    - a memory reference for the indirection

```
    call LL
LL: popl %ebx;            # ebx contains the current PC
    addl $PROCOFF, %ebx   # ebx points to the GOT entry for proc
    call *(%ebx)          # call indirect through the GOT
```

Lazy Binding

- implemented with a comapact but somewhat complex interaction between 2 data structures
  - GOT (Global Offset Table)
  - PLT (Procedure Linkage Table)
- if an object module calls any functions that are defined in shared libraries then it has its own GOT and PLT
- GOT in `.data` section
- PLT in `.text` section

Vector addition and multiplication examples (1)

```
void addvec (int *x, int *y, int *z, int n)
{
  int i;

  for (i=0; i<n; i++)
    z[i] = x[i] + y[i];

}

void multvect (int *x, int *y, int *z, int n)
{
  int i;

  for (i=0; i<n; i++)
    z[i] = x[i] * y[i];
}
```

Vector addition and multiplication examples (2)

```
#include <stdio.h>
#include "vector.h"

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

int main ()
{
  addvec(x, y, z, 2);
  printf("z= (%d %d)\n", z[0], z[1]);
  return 0;
}
```

The Global Offset Table for the previous examples

Address	Entry	Contents	Description
08049674	GOT[0]	0804969c	address of .dynamic section
08049678	GOT[1]	4000a9f8	identifying info for the linker
0804967c	GOT[2]	4000596f	entry point in dynamic linker
08049680	GOT[3]	0804845c	address of p̃ushl̃in PLT[1] (printf)
08049684	GOT[4]	0804846a	address of p̃ushl̃in PLT[2] (addvec)

The Global Offset Table Example (1)

- | 08049674 | GOT[0] | 0804969c | address of .dynamic section |
  - contains the address of the `.dynamic` seqment
  - dynamic linker use this address to bind procedure addresses
  - such as the location of the symbol table and relocation information
- | 08049678 | GOT[1] | 4000a9f8 | identifying info for the linker |
  - contains information that defines the module
- | 0804967c | GOT[2] | 4000596f | entry point in dynamic linker |
  - contains an entry point into the lazy binding code of the dynamic linker

The Global Offset Table Example (2)

- each procedure
  - that is defined in a shared object
  - and called by `main2.o` gets an entry in the GOT
- starting from GOT[3]
- GOT entries for `printf` defined in `libc.so`

-Got entries for `addvec` defined in `libvector.so`

- | 08049680 | GOT[3] | 0804845c | address of `pushl` in PLT[1] (printf) |
- | 08049684 | GOT[4] | 0804846a | address of `pushl` in PLT[2] (addvec) |

Procedure Linkage Table

- PLT[0] : a special entry that jumps into dynamic linker
- each called procedure has an entry in the PLT
  starting at PLT[1]
- PLT[1] : `printf`
- PLT[2] : `addvec`
- initially, after the program has been dynamically linked begins executing
  procedure `printf` and `addvec` are bound to the first instruction in their respective PLT entries

Procedure Linkage Table for the previous examples

```
PLT[0]
08048444:          pushl    0x8049678     # push &GOT[1]
 804844a:          jmp     *0x804967c     # jmp to *GOT[2] (linker)
 8048450:                                 # padding
 8048452:                                 # padding

PLT[1] <printf>
 8048454:          jmp     *0x8049680     # jmp to *GOT[3]
 804845a:          pushl   $0x0           # ID for printf
 804845f:          jmp      0x8048444     # jmp to PLT[0]

PLT[2] <addvec>
 8048464:          jmp     *0x8049684     # jmp to *GOT[4]
 804846a:          pushl   $0x8           # ID for addvec
 804846f:          jmp      0x8048444     # jmp to PLT[0]
```

GOT and PLT for addvec (1)

```
+----------+---------+----------+----------------------------------------+
|  Address | Entry   | Contents | Description                            |
+----------+---------+----------+----------------------------------------+
| 08049674 | GOT[ 0] | 0804969c | address of .dynamic section            |
| 08049678 | GOT[ 1] | 4000a9f8 | identifying info for the linker        |
| 0804967c | GOT[ 2] | 4000596f | entry point in dynamic linker          |
| 08049680 | GOT[ 3] | 0804845c | address of pushl in PLT[ 1] (printf)   |
| 08049684 | GOT[ 4] | 0804846a | address of pushl in PLT[ 2] (addvec)   |
+----------+---------+----------+----------------------------------------+
```

```
PLT[0]
08048444:         pushl      0x8049678     # push &GOT[1]
 804844a:         jmp       *0x804967c     # jmp to *GOT[2] (linker)
 8048450:                                  # padding
 8048452:                                  # padding

...               ...      ...             ...

PLT[2] <addvec>
 8048464:         jmp       *0x8049684     # jmp to *GOT[4]
 804846a:         pushl     $0x8           # ID for addvec
 804846f:         jmp        0x8048444     # jmp to PLT[0]
```

GOT and PLT for addvec (2)

- | 08049674 | GOT[0] | 0804969c | address of .dynamic section |
- | 08049684 | GOT[4] | 0804846a | address of pushl in PLT[2] (addvec) |
- 8048464: jmp *0x8049684 # jmp to *GOT[4]
- 804846a: pushl $0x8 # ID for addvec
- 804846f: jmp 0x8048444 # jmp to PLT[0]

Procedure Linkage Table Example (1)

- the call to addvec has the following form

  ```
  80485bb: e8 a4 fe ff ff      call 8048464 <addvec>
  ```

- at the first call, control is passed to the 1st instruction in PLT[2] which does the indirect jump through GOT[4]
- initially, each GOT entry contains the address of the push1 entry in the corresponding PLT engtry
- the indirect jump in the PLT simply transfers control back to the next instruction in PLT[2]
- this instruction pushes an ID for the addvec symbol onto the stack

Procedure Linkage Table Example (2)

- the last instruction jumps to PLT[0],
  which pushes another word of identifying information
  on the stack from GOT[1],
- then, jumps into the dynamic linker indirectly through GOT[2].
- the dynamic linker uses the two stack entries
  to determinethe location of `addvec`,
  overwrites GOT[4] with this address
  and passes control to `addvec`
- the only additional overhead from this point on
  is the memory reference for the indirect jump