Link 9.A Position Independent Code

Young W. Lim

2019-03-04 Mon

Outline

Based on

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

Compling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

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

TOC: PIC Data References

1. Accessing global variables
2. Global Offset Table (GOT)
3. each GOT entry is relocated
4. indirect reference through the GOT
5. indirect reference - a pattern of codes
6. global variable access using the GOT

Accessing global variables (1)

- PIC references to global variables
  based on the following fact
  - the data segment is always allocated *immediately after* the code segment
  - regardless of the memory location where
    an object module or a shared library is located
-
```
+--------------------+------------------+-----------------------+
| Read/Write segment | higher addresses | .data, .bss           |
+--------------------+------------------+-----------------------+
| Read-only segment  | lower addresses  | .text, .init, .rodata |
+--------------------+------------------+-----------------------+
starting from 0x08048000
```

Accessing global variables (2)

- the distance between
  - any instruction in the code (`.text`) segment and
  - any variable inthe data (`.data`) segment

  is a run-time constant

- independent of the absolute memory locations of code and data segments

- Global Offset Table (GOT) located at the beginning of `.data` segment

- `.data` follows `.text` segment

Global Offset Table (GOT)

- each global data object
  - a global variable
  - a function name

  has an entry in the GOT

- each entry contains the appropriate absolute address

- *each* object module has *its own* GOT,
  which references any global data

- the GOT is located at the beginning of `.data` segment

each GOT entry is relocated

- each entry in the GOT
  has a relocation record (relocation table)
- at load time, the dynamic linker relocates
  each entry in the GOT (of a global object)
- each entry contains the appropriate absolute address
- *each* object module has *its own* GOT
  that references global data

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

indirect reference - 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
```

- ```
        PC+$VAROFF        -- the address of the GOT entry for var
     M[ PC+$VAROFF ]      -- the absolute address of var
  M[ M[ PC+$VAROFF ] ]    -- the value of var
  ```
  ```

a pattern of codes - call LL, LL: popl %ebs

```
    call LL
LL: popl %ebx;           # ebx contains the current PC
```

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

a pattern of codes - the address LL

```
    call LL
LL: popl %ebx;          # ebx contains the current PC
```

- the LL address
  - the return address stored on the stack
  - the address of the popl instruction
  - the address to be popped by the popl instruction
  - the current PC to be stored into %ebx

a pattern of codes - `addl $VAROFF, %ebx`

```
addl $VAROFF, %ebx    # ebx points to the GOT entry for var
movl (%ebx), %eax     # fetch the absolute address to %eax
movl (%eax), %eax     # fetch the global variable
```

- `addl` adds a constant offest `$VAROFF` to `%ebx` for the appropriate entry in the GOT where the absolute address of a symbol can be fetched
  - initial `%ebx` : the current PC of the `popl %ebx`
  - final `%ebx` : PC + \$VAROFF
- now, the global variable can be accessed *indirectly* through the GOT entry contained in `%ebx`

a pattern of codes - `movl (%ebx), %eax, movl (%eax), %eax`

```
addl $VAROFF, %ebx   # ebx points to the GOT entry for var
movl (%ebx), %eax    # fetch the absolute address to %eax
movl (%eax), %eax    # fetch the global variable
```

- the 2 `movl` load the contents of the global variable indirectly through the GOD into register %eax
- the 1st `movl (%ebx), %eax` fetches the absolute address of a gloabl variable into %eax
- the 2nd `movl (%eax), %eax` the value of the global variable into %eax

TOC: PIC Function Calls

resolving external procedure calls (1)

- if the same approach would be ued
  as the PIC global variable references
  - this approach require 3 additional instructions
- instead of this approach, the lazy binding technique is used

```
    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
```

resolving external procedure calls (2)

- 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
  - but for subsequent calls
    - just one additional instruction
    - an indirect memory reference

Lazy Binding Technique

- implemented with 2 data structures : GOT and PLT
  - Global Offset Table
  - Procedure Linkage Table
- if an object module calls any functions of shared libraries then the object module has its own GOT and PLT
- GOT in `.data` section (absolute address)
- PLT in `.text` section

resolving a function address using PLT (1)

- The caller of a function in a different shared object transfers control to the start of the PLT entry associated with the function.
- The first part of the PLT entry *loads* the address from the GOT entry associated with the function to be called.
- The control is transferred to the code at this address.
- If the function has already been called at least *once*, or lazy binding is not used, then the address found in the GOT is the address of the function.

http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html#PLTEX

resolving a function address using PLT (2)

- If a function has never been called
  and lazy binding is used then the address
  in the GOT points to the second part of the PLT
- The second part *loads* the offset
  in the symbol table associated with the callee
- Control is then transferred to the special
  first entry of the PLT (PLT[0])
- This first entry of the PLT entry *calls*
  the dynamic linker giving it
  the offset into the symbol table
  and the address of a structure
  that *identifies* the location of the caller

http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html#PLTEX

resolving a function address using PLT (3)

- The dynamic linker finds
  the real address of the symbol.
  It will *store* this address
  in the GOT entry of the function
  in the object code of the caller
  and it will then transfer control to the function.

- Subsequent calls to the function
  from this object will find the resolved address
  in the first half of the PLT entry
  and will transfer control directly
  without invoking the dynamic linker

http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html#PLTEX

addvec and multvec

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

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

main

```
#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;
}
```

- in `main` function, `addvec` and `printf` functions are called

the example GOT entries

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]	0804845a	address of pushl in PLT[1] (printf)
08049684	GOT[4]	0804846a	address of pushl in PLT[2] (addvec)

GOT[0], GOT[1], GOT[2]

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 object module of interest

0804967c	GOT[2]	4000596f	entry point in dynamic linker

  - contains an entry point into the lazy binding code of the dynamic linker

GOT[3], GOT[4]

- each procedure
  - defined in a shared object
  - called by `main` gets an entry in the GOT
- starting from GOT[3]

08049680	GOT[3]	0804845a	address of `pushl` in PLT[1] (printf)

  - GOT entry for `addvec` defined in `libvector.so`

08049684	GOT[4]	0804846a	address of `pushl` in PLT[2] (addvec

  - GOT entry for `printf` defined in `libc.so`

the example PLT entries

PLT[0]			
8048444:	pushl	0x8049678	push addr(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]

the 1st `jmp` instructions of `PLT[ 0]`, `PLT[ 1]`, `PLT[ 2]`

- PLT[0] has a special entry that jumps into `*GOT[ 2]` (dynamic linker)
- PLT[1] has a `printf` entry that jumps into `*GOT[ 3]`
- PLT[2] has a `addvec` entry that jumps into `*GOT[ 4]`

- each called procedure (`printf`, `addvec`) has an entry in the PLT starting at PLT[1]

initial procedure address binding

- after the program has been dynamically linked and is ready to begin executing
  - `printf` procedure addresses are bound to the address of the first instruction in PLT[1] that is `jmp to *GOT[ 3]`

8048454: jmp *0x8049680 jmp to *GOT[3]

  - `addvec` procedure addresses are bound to the address of the first instruction in PLT[2] that is `jmp to *GOT[ 4]`

8048464: jmp *0x8049684 jmp to *GOT[4]

initial *GOT[3] and *GOT[4] GOT entry contents

Address	Entry	Contents	Description
08049680	GOT[3]	0804845a	address of pushl in PLT[1] (printf)
08049684	GOT[4]	0804846a	address of pushl in PLT[2] (addvec)

- `*GOT[ 3] = 0804845a` : `pushl $0x0 (printf id)`
  the address of the second instruction in PLT[1]
- `*GOT[ 4] = 0804846a` : `pushl $0x8 (addvec id)`
  the address of the second instruction in PLT[2]

- in PLT[1]

804845a	pushl	$0x0	ID for printf

- in PLT[2]

804846a:	pushl	$0x8	ID for addvec

initial jmp to *GOT[3] and jmp to *GOT[4]

- jmp to *GOT[3] ≡ jmp to M[8049680] ≡ jmp to 804845a
  the next instruction (pushl) of the 1st instruciton jmp to *GOT[3]
- jmp to *GOT[4] ≡ jmp to M[8049684] ≡ jmp to 804846a next instruction (pushl)
  the next instruction (pushl) of the 1st instruction jmp to *GOT[4]

PLT[1]	<printf>	8048454	jmp	*0x8049680	jmp to *GOT[3]
		804845a	pushl	$0x0	ID for printf
PLT[2]	<addvec>	8048464	jmp	*0x8049684	jmp to *GOT[4]
		804846a	pushl	$0x8	ID for addvec

Address	Entry	Contents	Description
08049680	GOT[3]	0804845a	address of pushl in PLT[1] (printf)
08049684	GOT[4]	0804846a	address of pushl in PLT[2] (addvec)

initial indirect jump in PLT

- initially, each GOT entry (`*GOT[ 3]`, `*GOT[ 4]`) contains the address of the `pushl` entry (the second instruction) in the corresponding PLT entry
- the indirect jump in the PLT (`jmp to *GOT[ 3]`, `jmp to *GOT[ 4]`) simply transfers control back to the next instruction in the PLT entry
- ID for `printf` : 0x0 on the stack (`pushl $0x0`)
- ID for `addvec` : 0x8 on the stack (`pushl $0x8`)

the last jmp to PLT[0] instruction of PLT entry contents

- the last instruction of PLT[1] and PLT[2] jumps to PLT[0]

PLT[1]	804845f:	jmp	0x8048444	jmp to PLT[0]
PLT[2]	804846f:	jmp	0x8048444	jmp to PLT[0]

PLT[0]	8048444: 804844a:	pushl jmp	0x8049678 *0x804967c	push addr(GOT[1]) jmp to *GOT[2] (linker)

- the first instruction of PLT[0]
  `pushl 08049678` = `pushl &GOT[ 1]`
- the second instruction of PLT[0]
  `jmp to 4000596f` = `jmp to *GOT[ 2]`

the instructions of PLT[0]

- `pushl 08049678` = `pushl &GOT[ 1]`
  pushes `$0x400a9f8` on the stack
  (identifying info for the linker)
- `jmp to 4000596f` = `jmp to *GOT[ 2]`
  jump into the dynamic linker indirectly
  to `*GOT[ 2]` = `4000596f`
  (entry point in dynamic linker)

PLT[0]	8048444:	pushl	0x8049678	push addr(GOT[1])
	804844a:	jmp	*0x804967c	jmp to *GOT[2] (linker)

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

two stack entries

- in PLT[1] and PLT[2]
  - at `*GOT[ 3] = 0804845a : pushl $0x0 (printf id)`
  - at `*GOT[ 4] = 0804846a : pushl $0x8 (addvec id)`

804845a:	pushl	$0x0	ID for printf
804846a:	pushl	$0x8	ID for addvec

- in PLT[0]
  - *push* `$0x400a9f8` on the stack
    `pushl &GOT[ 1] = pushl 08049678`

8048444:	pushl	0x8049678	push addr(GOT[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

determing the address of printf and addvec

- *GOT[3] = 0804845a : pushl $0x0 (printf id)
- *GOT[4] = 0804846a : pushl $0x8 (addvec id)
- in PLT[0]
  - *push* $0x400a9f8 on the stack
    pushl &GOT[1] = pushl 08049678
- the dynamic linker uses the two stack entries
  to determine the actual locations of printf and addvec

update *GOT[3] and *GOT[4]

- intially
  - `*GOT[ 3] = 0804845a` : `pushl $0x0 (printf id)`
  - `*GOT[ 4] = 0804846a` : `pushl $0x8 (addvec id)`
- finally
  - `*GOT[ 3] = <printf> address`
  - `*GOT[ 4] = <addvec> address`

- the dynamic linker overwrites GOT[3] and GOT[4] with these newly determined addresses and passes control to `printf` or `addvec`
- the only additional overhead from this point on is the memory reference for the indirect jump

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]

steps of call to <addvec> (1)

- the call to addvec
  `80485bb: e8 a4 fe ff ff      call 8048464 <addvec>`
- 8048464 is the address of addvec entry of PLT[2]

8048464:	jmp	*0x8049684	jmp to *GOT[4]

- at the first call to addvec,
  control is passed to the 1st instruction in PLT[2]

- the indirect `jmp to *GOT[ 4]` tranfers control back
  to the next instruction `pushl $0x8`

804846a:	pushl	$0x8	ID for addvec

- this instruction pushes an ID 0x8
  for the addvec symbol onto the stack

steps of call to <addvec> (2)

- the last instruction jumps to PLT[0],
  which pushes another word of identifying information
  `$0x400a9f8` on the stack from GOT[1]

- in PLT[0]
  - *push* `$0x400a9f8` on the stack
    `pushl &GOT[ 1] = pushl 08049678`

8048444	pushl	0x8049678	push addr(GOT[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

steps of call to <addvec> (3)

- then, jumps into the dynamic linker indirectly through GOT[2].

804844a	jmp	*0x804967c	jmp to *GOT[2] (linker)

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

steps of call to <addvec> (4)

- the dynamic linker uses the two stack entries
  to determinethe location of `addvec`,
  overwrites GOT[4] with this address
  and passes control to `addvec`

Address	Entry	Contents	Description
08049684	GOT[4]	0804846a	address of pushl in PLT[2] (addvec)

08049684	GOT[4]	....	actual address of addvec