ELF1 7B PIC Method - ELF Study 1999

Young W. Lim

2020-04-17 Fri

Outline

1. Relocs and memory locations
   - TOC
   - Overview
   - Code and data segments
   - ELF Relocations
   - Global and local symbol relocs
2. PIC mechanism
   - TOC
   - Operations in the code
   - Operations in the PLT
   - OPerations in the GOT
3. PIC's accessing absolute addresses
   - TOC
   - 1. Global Offset Table Addressing
   - 2. Procedure Linkage Table Addressing

TOC: Relocs and memory locations

Handling inter-related referece

- linking in the old days
  - at compile time, inter-related references are not resolved
  - .o files include a reloc object that contains
    the information on these inter-related references
  - at link time, the linker would merge these informations
    in .o files building a table of where symbols are ultimately located.
  - the linker would run through the set of relocs, filling them in

`http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`

Reloc attributes

- A reloc consists of three parts:
  - where in memory the fix is to be made
  - the symbol which is involved in the fix
  - an algorithm that the linker should use to create the fixup
- The algorithm can be as simple as `R_386_32`
  "use the symbol memory location; store it in binary"
- complicated, such as `R_ARM_PC26`
  "calculate the distance from here to the symbol, divide by 4, subtract 2 and add the result to the 3 lower bytes"

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Static linking

- these relocs are scattered through the .o files,
  and are used at link time create the correct binary executable file.
- resolving all the relocs is necessary
- in the days of static linking

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Dynamic linking

- run-time linking
  the designers of the ELF format enabled reloc entites
  to hold run-time resolution information.
- So now executable files may have relocs in them,
  even after linking
  - ELF implements run time linking
    by deferring function resolution
    until the function is called.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

New algorithm

- However, new algorithms are required to inform
  how these fixups are to be done.
- Hence the introduction of a new family of reloc algorithms

`http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`

Relocation

- Binary executables often need certain bits of information *fixed up* before they execute
- ELF binaries carry a list of relocs (relocation table) which describe these *fixups*

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Relocation entries

- Each reloc contains relocation entry
  - the address in the binary that is to get the *fixup* (offset)
  - the algorithm to calculate the fixup (type)
  - a symbol (string and object length)
- At *fixup* time,
  the algorithm (type) uses the offset & symbol,
  along with the value (addend) currently in the file,
  to calculate a new value to be stored into memory.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Code and Data

- One of the characteristics of the ELF binary system is a separation of code and data.
- The code of apps and libraries is marked read-only and executable
- The data is marked read-write, and not-executable.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Code segment (1)

- The code is read-only
  so that multiple processes can share the code,
  - the code is loaded into memory only once.
  - the code is never modified,
    and appears identical in each process space.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Code segment (2)

- Each process has its own page tables,
  mapping the code into its own memory.
  - therefore the code must be position independent
  - each process can load the code into a different address
- The code segment is allowed to contain
  constant pointers and strings (.rodata).

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Data segment (1)

- The data segment is read-write and
  is mapped into each process space differently.
- In Linux, each data segment is loaded
  from the same base mmap (identical),
  but it is marked copy-on-write (own copy later)

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Data segment (2)

- after the first write,
  each process has its own copy of the data.
  (in its own read-write segment)

- therefore, relocs can only point
  to the data segment (_identically_)

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ELF code and data segments memory layout

- an ELF executable consists of a group of code segments followed by a group of data segments
- GOT is located at the beginning of data pages
- regardless of the load address (wherein the address space the program loaded) the offset from the code to the data doesn't change

J. R. Levine, Linkers and Loaders

ELF data references

- if the code can load its own address into a register,
  the data will be at a known distance from that address
- references to data in the program's own data segment can use efficient based addressing with fixed offsets

`J. R. Levine, Linkers and Loaders`

Relocs in code and data segments

- the relocs in the data segment are *easy* to be done
  - add relative offsets or
  - write absolute addresses
- the relocs in the code area are more *difficult*.
  - the ELF reloc design makes the code relocs *intact*
  - an GOT entry in the data area is to be filled, (Global Offset Table).

`http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`

Relocs using GOT for a global object

- if code needs to refer to a global object,
  it refers to an entry in the `GOT[]`,
  - at run-time, the `GOT` entry is *fixed-up*
    to point to the correct address of the global object.
  - the code space need never be *fixed-up* at run time.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Relocs using GOT for a local object

- if the code needs to refer to a local object, it refers to it relative to the `&GOT[ 0]`;
  - no run-time *fixed-up*
  - this too is position independent

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Function addresses

- References to a function address from an executable file and from the shared objects associated with the file must resolve to the same value.
- References from within shared objects
  - will normally be resolved (by the dynamic linker) to the virtual address of the function itself
- References from within the executable file to a function defined in a shared object
  - will normally be resolved (by the linkage editor) to the address of the PLT entry for that function within the executable file.

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

Relocs using PLT

- If the code needs to jump to a subroutine
  in a different module,
  the linker creates an array of *jump-stubs*
  called the PLT (procedure linkup table)
- these *jump-stubs* in the PLT jump indirect,
  using an entry in the GOT[]
  to implement the far call

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Deferring function resolution

- ELF implements run time linking
  by deferring function resolution
  until the function is called.

- calls to library functions go through a *fix-up* process
  just after the first time call is made

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

GOT relative

- GOT-relative (GOTOFF) code is made
  relative to the start of the GOT table (O)

- relative to the load address of the module (X)

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Global symbol relocs

- global relocs must neccessarily involve
  the three aspects of a reloc:
  - where in memory the reloc is to be made
  - the symbol involved in the reloc
  - the algorithm used to make the fixup.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Local symbol relocs

- a local symbol can be fixed in memory with respect to a memory "section",
- the object file is allowed to drop the local symbol name, and replace it with a section-plus-offset

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ARM code example (1)

```
        .section .text
        mov     r0, r0   @sample code
.L2:    call    _do_something
        ldr     r6, .L3  @this code need a reloc!
        mov     r0, r0
.L4:    .word   Lextern
.L3:    .word   .L2      @this read-only data needs a reloc
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ARM code example (2)

- the code on the 3rd line (the call) needs to be fixed up, but that's easy, since it's a PC relative fixup.

```
.L2:    call   _do_something
```

- If the .o file has no idea where .Lextern is,

```
.L4:    .word  Lextern
```

  it must neccessarily create a reloc which refers to symbol Lextern

```
.L4:    .word  0  R_ARM_32 Lextern
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ARM code example (3)

- the word at .L3 needs a fixup as well.

```
.L3:     .word  .L2      @this read-only data needs a reloc
```

- If the .o file can determine the location of a local symbol, such as L2, then it is allowed to replace the symbol with a section-plus-offset
- The offset is stored in the reloc target address, and the section is an entry in the reloc symbol table

```
.L3:     .word  4  R_ARM_32 .text
```

- This reduces the number of symbols in the symbol table, making run-time linking easier.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ARM code example (4)

- the R_*_GOTOFF and R_*_GOT32 relocs include
  - R_386_GOTOFF : GOT-relative, local symbol address
  - R_386_GOT32 : GOT-relative, GOT entry address

  an offset from &GOT[0], which is usually about
  halfway through the module.
- The R_ARM_RELATIVE relocs, on the other hand,
  contains an offset from the beginning of the module. Tradition.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

TOC: PIC mechanism

- Operations in the code
- Operations in the PLT
- Operations in the GOT

TOC: Operations in the code

Lazy binding and constraints

- ELF dynamic linking defers the resolution of jump / call addresses until the last minute.
- Constraints:
  - should not force a change in the assembly code produced for apps but may cause changes as an assembly code is changed for PIC
  - all executable codes must not be modified at run time any modified data must not be executed at run time

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Three steps in a far jump

1. start in the code
2. go through the PLT
3. using a pointer from the GOT

- the GOT entries that are used for PLT execution have default addresses initially
- give control back to the corresponding PLT entry stub
- consisitng of `push` and `jmp PLT[0]` sequence

`http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`

Operations in the code

```
call function_call_n
```

- the *relative* jump or call
- the target is a PLT entry `PLT[n+1]`
  - it is *(n+1)-th* entry not the *n-th* entry
  - `PLT[0]` is the special first entry
- `call PLT[n+1]` : similar to a normal call
- assume *n* is a number

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

TOC: Operations in the PLT

(1) PLT entry : stub code

- the PLT is a synthetic area, created by the linker
- exists in both executable and libraries
- an array of stubs, one per imported function call
- through PLT[0], the resolver is called at last

```
PLT[n+1]: jmp     *GOT[n+3]
          push    #n            ; push n as a argument to the resolver
          jmp     PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(2) indirect call through GOT

- a call to PLT[n+1] will result in *indirect call* through GOT[n+3]
  - because of three special GOT entries : GOT[0,1,2]

    ```
    jmp *GOT[n+3]    ; 6-byte long
    ```

- initially, the value at GOT[n+3] points back to PLT[n+1]+6
  - the next instruction after the 6 byte instruction jmp *GOT[n+3]
  - push and jmp PLT[0] sequence

```
PLT[n+1]:     jmp     *GOT[n+3]     ; 6 bytes insturction
PLT[n+1]+6:   push    #n            ; push n as a argument to the resolver
              jmp     PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(3) push, jmp PLT[0] sequence

- by the instruction at PLT[n+1]+6, *n* is pushed onto the stack as an argument for the resolver (push #n)
- consider *n* as an *ID* for the called library function
- the resolver uses the argument *n* on the stack in resolving the symbol *n* (here *n* is treated as a symbol)

```
PLT[n+1]:     jmp    *GOT[n+3]     ; 6 bytes insturction
PLT[n+1]+6:   push   #n            ; push n as a argument to the resolver
              jmp    PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(4) overriding the default GOT[n+3]

- the resolver is called by the stub at PLT[0]
- the resolver modifies the default value at GOT[n+3]
  to point the correct target symbol *n*
- overrides PLT[n+1]+6 (the default value at G[n+3])

- thus after the first call, the control is taken
  directly to the correct target symbol *n* (function_call_n)
  instead of executing the push-jump sequence (through

```
PLT[n+1]:     jmp    *GOT[n+3]     ; 6 bytes insturction
PLT[n+1]+6:   push   #n            ; push n as a argument to the resolver
              jmp    PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(5) the special entry PLT[0]

- the resolver needs 2 argument
  - symbol *n* is already on the stack
  - pointer to the relocation table : GOT[1]
  - &GOT[1] is added on the stack
- the resolver that is located in ld-linux.so.2 can determine *which library function* is asked for its service using these two arguments on the stack
- GOT[2] : entry point of dynamic linker

```
PLT[0]:    push    &GOT[1]
           jmp     GOT[2]          ; entry point of dynamic linker
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Summary of steps

```
1. call PLT[n+1]

2. jmp *GOT[n+3]

   at the 1st call, jmp PLT[n+1]+6

   - push #n
   - jmp PLT[0]

     - push &GOT[1] (pointer to the reloc table)
     - jmp GOT[2] (entry point of dynamic linker)

   after the 1st call, jmp n
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

TOC: Operations in the GOT

(1) three types of GOT entries

- the GOT contains *helper pointers*
  for both PLT fixups and GOT fixups
  - the first 3 entries are special and reserved
  - the next M entries belong to the PLT fixups
  - the next D entries belong to various data fixups
- the GOT is a synthetic area, createdy by the linker
  exists in both executables and libraries
  - each library and executable
    gets its own PLT and GOT array

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(2) the three special GOT entries

- the special 3 entries in the GOT
- GOT[0] : linked list pointer used by the dynamic linker
  address of .dynamic section
- GOT[1] : pointer to the reloc table for this module
  module identification info for the linker
- GOT[2] : pointer to the fixup / resolver code, located in ld-linux.so.2
  entry point in dynamic linker

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(3) the PLT-fixup

- when the GOT is first set up,
  all the GOT entries related to PLT fixups
  are pointing to code back at PLT[0]
- GOT[n+3] are pointing back to PLT[n+1]+6
  which eventually jump to PLT[0] to call the resolver

```
PLT[n+1]: jmp     *GOT[n+3]
          push    #n              ; push n as a argument to the resolver
          jmp     PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(4) the PLT-fixup vs data-fixup

- M entries belong to the PLT fixups

GOT[3]	indirect function call helpers
GOT[4]	indirect function call helpers
...	...
GOT[3+M-1]	indirect function call helpers,
	one per imported function

- D entries belong to various data fixups

GOT[3+M]	indirect pointers to global data references
GOT[3+M+1]	indirect pointers to global data references
...	...
GOT[end]	indirect pointers to global data references

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

TOC: PIC's accessing absolute addresses

- Position independent references to absolute locations
- Loadable object file
- The link editor and the runtime linker
- Separate GOT's
- Address of the dynamic structure
- Memory segment addresses
- `_GLOBAL_OFFSET_TABLE_`

Position independent references to absolute locations

- PIC cannot, in general, contain absolute virtual addresses
- GOTs hold absolute addresses in private data
- Addresses are therefore available without compromising
  the position-independence and
  shareability of a program's text.
- A program references its GOT in a position-independent way
  and extracts absolute values.
- this technique redirects position-independent references
  to absolute locations.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-74186/index.html

Loadable object file

- Initially, the GOT holds information
  as required by its relocation entries.
- After the system creates memory segments
  for a loadable object file,
  the runtime linker processes the relocation entries
  for example, `R_386_GLOB_DAT` in the entries of GOT

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-74186/index.html

The link editor and the runtime linker

- The runtime linker
  determines the associated symbol values,
  calculates their absolute addresses, and
  sets the appropriate memory table entries to the proper values.
- Although the absolute addresses are unknown
  when the link-editor creates an object file,
  the runtime linker
  knows the addresses of all memory segments and
  can thus calculate the absolute addresses
  of the symbols contained therein.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-74186/index.html

Separate GOT's

- If a program requires direct access
  to the absolute address of a symbol,
  that symbol will have a GOT entry.
- Because the executable file and shared objects
  have separate global offset tables,
  a symbol's address can appear in several tables.
- The runtime linker processes all the GOT relocations
  before giving control to any code in the process image.
- This processing ensures that
  absolute addresses are available during execution.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-74186/index.html

Address of the dynamic structure

- The table's entry zero is reserved
  to hold the address of the dynamic structure,
  referenced with the symbol `_DYNAMIC`.
- This symbol enables a program, such as the runtime linker,
  to find its own dynamic structure
  without processing its relocation entries.
- This method is especially important for the runtime linker,
  because it must initialize itself without relying
  on other programs to relocate its memory image.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-74186/index.html

Memory segment addresses

- The system can choose different memory segment addresses for the same shared object in different programs.
- It can even choose different memory segment addresses for different executions of the same program.
- Nonetheless, memory segments do not change addresses once the process image is established.
- As long as a process exists, its memory segments reside at fixed virtual addresses.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-74186/index.html

_GLOBAL_OFFSET_TABLE_

- A GOT's format and interpretation are processor-specific
- For SPARC and IA processors, the symbol
  GLOBAL_OFFSET_TABLE
  can be used to access the table.
- this symbol can reside in the middle of the .got section, allowing both negative and nonnegative subscripts into the array of addresses.
- The symbol type is an array of Elf32_Addr for 32-bit code, and an array of Elf64_Addr for64-bit code:

```
extern  Elf32_Addr  _GLOBAL_OFFSET_TABLE_[];
extern  Elf64_Addr  _GLOBAL_OFFSET_TABLE_[];
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-74186/index.html

TOC: 2. Procedure Linkage Table Addressing (1)

- GOT and PLT for absolute locations
- Function calls and link-editor
- Function calls and runtime linker
- Procedure Linkage Table example
- Different addressing modes

TOC: 2. Procedure Linkage Table Addressing (2)

- Step 1 - the second and third entries in the GOT
- Step 2 - the GOT address in `%ebx`
- Step 3 - jump to the corresponding PLT entry
- Step 4 - the first instruction of the PLT entry
- Step 5 - the seond instruction of the PLT entry
- Step 6 - the first entry of the PLT
- Step 7 - the actual address of the function
- Step 8 - the subsequent function calls
- `LD_BIND_NOW`

GOT and PLT for absolute locations

- The GOT converts *position-independent* address calculations to absolute locations.
- The PLT converts *position-independent* function calls to absolute locations.
- Executable files and shared object files have *separate* GOT's and *separate* PLT's

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Function calls and link-editor

- The link-editor cannot resolve execution transfers such as function calls *from* one executable or shared object *to* another
- So, the link-editor arranges to have the program transfer control to entries in the PLT.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Function calls and runtime linker

- For 32-bit IA dynamic objects,
  the PLT *resides* in shared text but
  uses addresses in the private GOT.
- The runtime linker determines
  the absolute addresses of the destinations and
  modifies the GOT's memory image accordingly.
- The PLT thus redirects the entries
  without compromising the position-independence and
  shareability of the program's text.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Procedure Linkage Table Example

```
.PLT0:                              .PLT0:
    pushl   got_plus_4                  pushl   4(%ebx)
    jmp     *got_plus_8                 jmp     *8(%ebx)
    nop;    nop                         nop;    nop
    nop;    nop                         nop;    nop
.PLT1:                              .PLT1:
    jmp     *name1_in_GOT               jmp     *name1@GOT(%ebx)
    pushl   $offset                     pushl   $offset
    jmp     .PLT0@PC                    jmp     .PLT0@PC
.PLT2:                              .PLT2:
    jmp     *name2_in_GOT               jmp     *name2@GOT(%ebx)
    pushl   $offset                     pushl   $offset
    jmp     .PLT0@PC                    jmp     .PLT0@PC

-- Absolute code                    -- Position independent code
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Different addressing modes

- the PLT instructions use
  different operand addressing modes for
  - absolute code
  - position-independent code.

```
 got_plus_4                    4(%ebx)
*got_plus_8                   *8(%ebx)
*name1_in_GOT                 *name1@GOT(%ebx)
*name2_in_GOT                 *name2@GOT(%ebx)
-- Absolute code           -- Position independent code
```

- Nonetheless, their interfaces to the runtime linker
  are the same.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Step 1 - the second and third entries in the GOT

- the runtime linker and program cooperate to resolve the symbolic references through the PLT and the GOT
- When first creating the memory image of the program, the runtime linker sets the 2^{nd} and 3^{rd} entries in the GOT to special values.
  - the scond GOT entry : identifying information
  - the third GOT entry : jump to the runtime linker

```
   got_plus_4                 4(%ebx)
   got_plus_8                 8(%ebx)
-- Absolute code           -- Position independent code
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Step 2 - the GOT address in %ebx

- If the PLT is position-independent,
  the address of the GOT must be in `%ebx`.
- *each* shared object file in the process image
  has *its own* PLT
- control transfers to a PLT entry
  only from within *the same* object file
- So, the calling function must set
  the GOT base register `%ebx`
  before it calls the PLT entry

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Step 3 - jump to the corresponding PLT entry

- for example, the program calls `name1`, which transfers control to the label `.PLT1`.
  - R_386_PLT32 reloc in `.o` file has been resolved by PC-relative, PLT entry address from the symbol reference (function call)
  - R_386_JMP_SLOT reloc in `.so` file has a entry in GOT and the the runtime linker will fill it with the target address

```
.PLT1:
jmp      *name1_in_GOT
pushl    $offset
jmp      .PLT0@PC
```

```
.PLT1:
jmp      *name1@GOT(%ebx)
pushl    $offset
jmp      .PLT0@PC
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Step 4 - the first instruction of the PLT entry

- The first instruction (`jmp *name1@GOT(%ebx)`) jumps to the address in the GOT entry for `name1`.
- Initially, the GOT entry holds the address of the following instruction of `jmp`
  - the address of the 2^{nd} instruction of the PLT entry (`pushl $offset`)
  - not the real address of `name1`.

```
.PLT1:
jmp      *name1_in_GOT
pushl    $offset
jmp      .PLT0@PC
```

```
.PLT1:
jmp      *name1@GOT(%ebx)
pushl    $offset
jmp      .PLT0@PC
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Step 5 - the remaining instructions of the PLT entry

- The program pushes the offset of a relocation entry on the stack.
  a 32-bit, nonnegative byte offset in the relocation table
- The designated relocation entry has
  - R_386_JMP_SLOT relocation type
  - relocation offset specifies the GOT entry for name1
    used in the previous instruction jmp *name1@GOT(%ebx)
  - a symbol table index, which the runtime linker
    uses to get the referenced symbol, name1.
- After pushing the relocation offset, the jmp .PLT0@PC
  instruction jumps to .PLT0, the first entry in the PLT.

```
pushl   $offset                 pushl   $offset
jmp     .PLT0@PC                jmp     .PLT0@PC
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Step 6 - the first entry of the PLT

- The `pushl 4(%ebx)` instruction pushes
  the address of the second GOT entry on the stack,
  giving the runtime linker one word of identifying information

- then `jmp * 8(%ebx)` instruction jumps to
  the address in the third GOT entry,
  to jump to the runtime linker

```
pushl   got_plus_4                 pushl   4(%ebx)
jmp     *got_plus_8                jmp     *8(%ebx)
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Step 7 - the actual address of the function

- The runtime linker unwinds the stack,
  - checks the designated relocation entry
    - relocation type `R_386_JMP_SLOT`
    - relocation offset `*name1@GOT(%ebx)`
    - symbol table index
  - gets the symbol's value
  - stores the actual address of `name1` in its GOT entry,
  - jumps to the destination (the actual address of `name1`)

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Step 8 - the subsequent function calls

- Subsequent executions of the PLT entry transfer directly to `name1`, without calling the runtime linker again (the address in the third GOT entry)
- The `jmp` instruction at `.PLT1` jumps to `name1` instead of falling through to the `pushl` instruction.

```
.PLT1:                          .PLT1:
jmp      *name1_in_GOT          jmp      *name1@GOT(%ebx)
pushl    $offset                pushl    $offset
jmp      .PLT0@PC               jmp      .PLT0@PC

*name1_in_GOT                   *name1@GOT(%ebx)
initially contains the address of the next  pushl instruction
then is modified to have the address of the called function
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Initial function calls

- .PLT1 (the PLT entry for `name1`)

```
.PLT1:                            .PLT1:
    jmp     *name1_in_GOT             jmp    1P   *name1@GOT(%ebx)
    pushl   $offset                   pushl    $offset
    jmp     .PLT0@PC                  jmp      .PLT0@PC
```

- .PLT0 (the first PLT entry)

```
.PLT0:                             .PLT0:
    pushl   got_plus_4                 pushl    4(%ebx)
    jmp     *got_plus_8                jmp      *8(%ebx)
```

- runtime linker jumps to the address of `name1`

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Subsequent function calls

- .PLT1

```
.PLT1:                          .PLT1:
    jmp     *name1_in_GOT           jmp   1P   *name1@GOT(%ebx)
```

- directly jump to name1

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

LD_BIND_NOW_

- The LD_BIND_NOW environment variable changes dynamic linking behavior.
- If its value is non-null, the runtime linker processes R_386_JMP_SLOT relocation entries (PLT entries) before transferring control to the program.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Relocation Entry

- relocation entries for code are placed in `.rel.text`
- relocation entries for initialized data are in `.rel.data`
-
```
typedef struct {
    int offset;
    int symbol:24,
        type:8;
} Elf32_Rel;
```

  - The relocation entry for the PLT example has
    - relocation offset specifies the GOT entry for `name1` used in the previous instruction `jmp *name1@GOT(%ebx)`
    - symbol table index, which the runtime linker uses to get the referenced symbol, `name1`.
    - relocation type `R_386_JMP_SLOT`

https://people.cs.pitt.edu/~xianeizhang/notes/Linking.html#reloc

Offsets in a PLT entry

- Offset in a PLT entry
  - the pushed offset `$offset` $= 8n =$ `sizeof(Elf32_Rel)` $* n$
    - the offset of a relocation entry in the relocation table
  - the jump offset `var@GOT(%ebx)`
    - The symbol term (reference) is replaced with offset from the start of the GOT to the GOT slot for the symbol

```
.PLT1:
jmp      *name1_in_GOT
pushl    $offset
jmp      .PLT0@PC
```

```
.PLT1:
jmp      *name1@GOT(%ebx)
pushl    $offset
jmp      .PLT0@PC
```

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

Offset in R_386_JMP_SLOT type relocation

- relocation entries for a PLT are placed in `.rel.plt`
- Relocation offset of `R_386_JMP_SLOT` type relocation specifies the GOT entry for a given function (`.got.plt`)

- Its offset member gives the location of a *GOT entry*
- The runtime linker modifies the *GOT entry* to transfer control to the designated symbol address.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html

PLT (6)

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-1235/index.html