ELF Nightmares: GOTs, PLTs, and Relocations Oh My

John Baldwin

BSDCan

13 June 2025

Overview

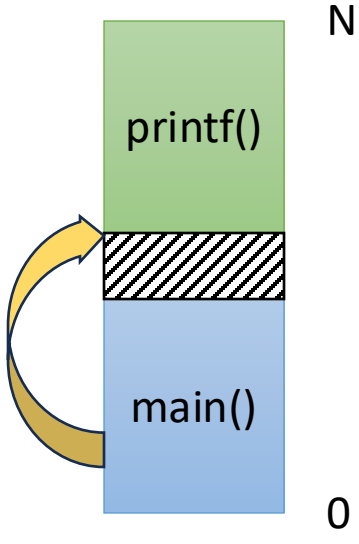
- The problem
- Relocations
- Code generation modes
- GOT indirection
- Function indirection via PLT
- Code generation revisited
- Copy Relocations and Canonical PLT Entries
- Indirect functions

The Problem: Mapping Symbols to Addresses

- Source code refers to functions and data objects by symbol names
- Machine code refers to functions and data by addresses

```
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return (0);
}
```



A few definitions (1)

- A **compiler** translates source code from one language into another
  - Often the second language is assembly
  - clang, gcc

- An **assembler** generates **object files** containing machine code and initial data values
  - Object files are generally split into sections where each section holds a single type of data
  - clang, as

A few definitions (2)

- A **static linker** merges sections from one or more object files into an output executable or shared library (**linked output file**)
  - Executables can be either static or dynamic
  - ld.lld, ld.bfd

- A **run-time loader** prepares a dynamic executable for execution including loading dependencies (shared libraries)
  - Also called dynamic linker, run-time linker
  - ld-elf.so.1

The Problem: Refined

- Converting source code into machine code is a multi-step process
- Machine code is generated by the compiler and assembler and stored in object files
- But the final addresses of symbols are not known until the executable is linked by the static linker
- Solution: The static linker patches generated machine code and data with the final addresses

- New problem: How to describe this patching?

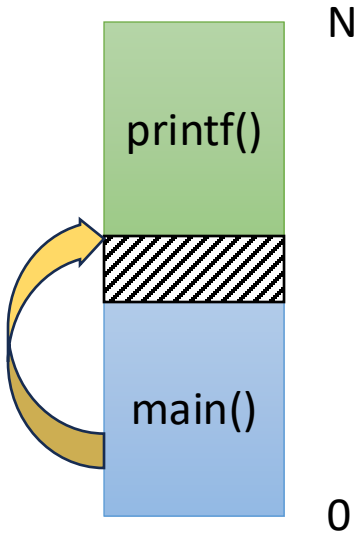
Linking a Static Executable

- Each object file is assembled to run at address zero
- As the static linker combines input sections from object files it rearranges input sections (e.g. merging all .text input sections into a single .text output section)
- Once the input sections are merged and laid out, the static linker can then compute the final addresses of all symbols
- Once the final addresses are known, the static linker can apply patches to instructions or data

Static Linking Hello World

- Target address of branch instruction from main() in hello.o patched to final address of printf() imported from libc.a

- Global variables that contain pointers must also be patched (e.g. TAILQ_HEAD_INITIALIZER)



Dynamic Linking

- When using shared libraries, some addresses are not known until run time
- These addresses cannot be patched by the static linker in the linker output file but are instead patched at run time by the run-time loader in the in-memory copy of an output file
- For example, if the hello world example is linked dynamically, the final address of printf() is not known until the shared C library is loaded into the virtual memory of a process

Relocations

- The process of rearranging object file input sections in the static linker **relocates** individual functions and global variables to new addresses
- **Static relocations** are data structures generated by compilers and assemblers to describe the patches required by object files and are consumed by the static linker
- **Dynamic relocations** are similar data structures generated by the static linker to describe patches required by linked output files and are consumed by the run-time loader
- In ELF, both types of relocations use the same underlying data structures

ELF Relocations

- ELF relocations contain fields that represent the following
  - The target address (file offset) to be patched
  - The type of relocation to perform
  - The symbol the target is referring to (optional)
  - An addend to add to the resolved address (optional)

- Elf_Rel does not include an explicit addend
  - If an addend is used, it is stored in the target address (relocation is not idempotent!)
- Elf_Rela does include an explicit addend

ELF Relocation Types

- Types are architecture-dependent
- Type specifies various properties of the relocation
  - Address calculation formula
  - How to store the result

- Some classes of relocation types
  - Absolute address (R_*_ABS*) (S + A)
  - Address relative to the object's base address (R_*_RELATIVE) (B + A)
  - Address relative to the current PC value (S + A - P)

- https://riscv-non-isa.github.io/riscv-elf-psabi-doc/#reloc-table

Multi-Relocation Patches

- Some architectures (e.g. AArch64 and RISC-V) use multiple instructions to construct an address
- Each instruction needs its own relocation

```
return (&foo);
```

When assembled b[illegible]es:

```
lui  a0, %hi(foo)
addi a0, a0, %lo(foo)
ret
```

Upper 20 bits of &foo

Low 12 bits of &foo

Examining Relocations

- `readelf -r`: List of relocations by section
- `objdump -r`: List of static relocations by section
- `objdump -R`: List of dynamic relocations by section
- `objdump -dr`: Display static relocations inline with disassembly
- Compiler explorer: https://godbolt.org

Compiler Explorer: RISC-V Example

```
extern int foo;

void *bar(void)
{
     return (&foo);
}
```

RISC-V rv64gc clang 20.1.0 -O2 -fno-pic

```
bar:
     lui a0, %hi(foo)
     addi a0, a0, %lo(foo)
     ret
```

Compiler Explorer: RISC-V Example

```
extern int foo;

void *bar(void)
{
     return (&foo);
}
```

RISC-V rv64gc clang 20.1.0 -O2 -fno-pic

- Compile to binary object

```
bar:
       lui a0,0x0
              R_RISCV_HI20 foo
              R_RISCV_RELAX *ABS*
       mv a0,a0
              R_RISCV_LO12_I foo
              R_RISCV_RELAX *ABS*
       ret
```

ELF Object File Code Generation

- Position-Dependent Executable (PDE)
  - Static executables and libraries
  - Dynamic executables
  - -fno-pic
- Position-Independent Code (PIC)
  - Shared libraries
  - -fpic / -fPIC
- Position-Independent Executable (PIE)
  - Static executables and libraries
  - Dynamic executables
  - -fpie / -fPIE

Evolution of ELF Object Files

- Static executables as the starting point / base case (PDE)
- Shared libraries built on top of this base
  - Library object files compiled as PIC and linked into a .so
  - Executable object files remain unchanged, but linked differently
- PIE added relatively recently
  - Motivated by ASLR
  - Run-time relocations like shared libraries
  - ”Optimized” PIC

PIC Mode Differences

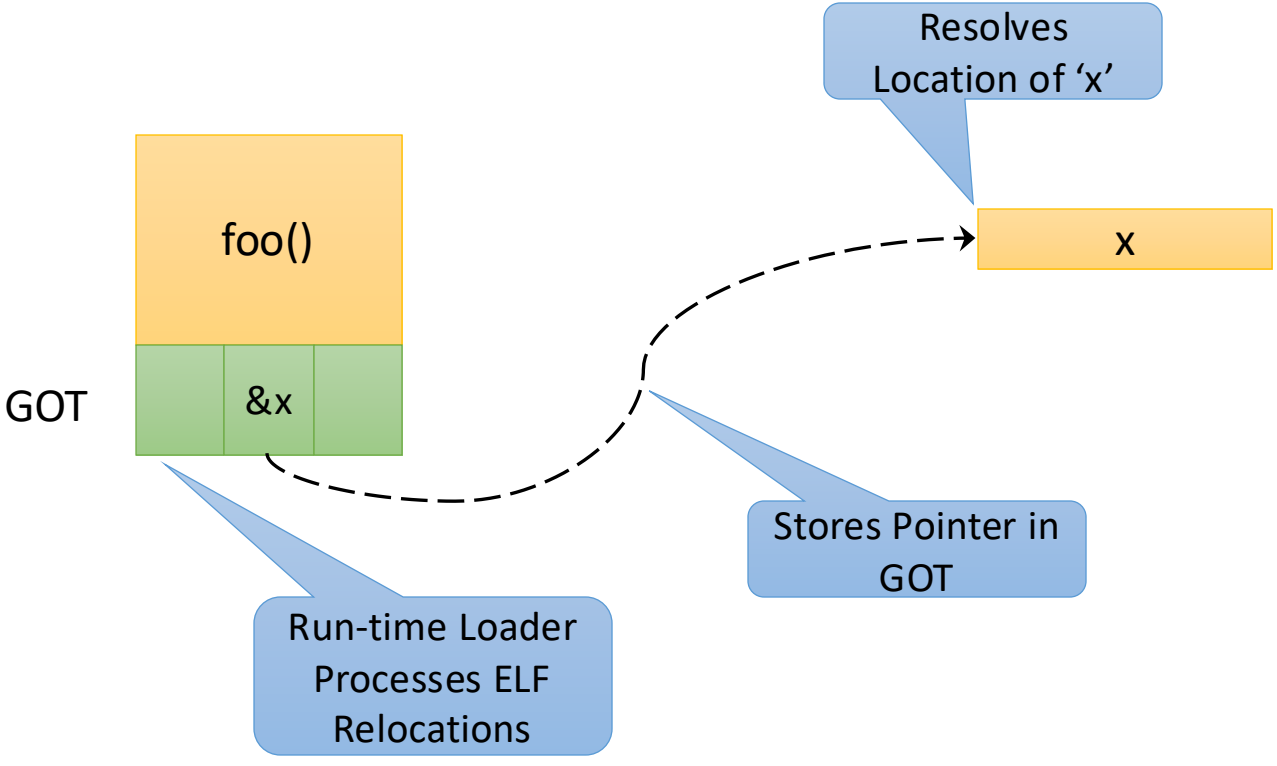
- Shared libraries can be loaded at different addresses
  - Symbols cannot be directly accessed at absolute addresses
  - PC-relative access to objects within the same shared library are ok
- Global variables and functions might be defined in a different linked output file
  - Address of variables and functions are only known at run time by the run-time loader

Accessing Globals in PIC

- Run-time loader could patch instructions directly similar to static linker
  - Prevents sharing underlying physical pages containing .text across multiple processes

- Instead, variables and functions are accessed indirectly
  - Static linker constructs an array of pointers known as a **Global Offset Table** (**GOT**)
  - Compiler generates code that reads pointers from GOT and dereferences them to access data
  - Static linker generates dynamic relocations describing how to populate GOT entries that are handled by the run-time loader

Accessing Globals in PIC



Preemptible Symbols (1)

- “Remote” symbols (those not defined in the same linker output file) must always be accessed indirectly via GOT entries in PIC mode
  - Note that the compiler does not know which “remote” symbols in a single object file will end up “remote” in the final linker output file

- However, ELF also uses GOT indirection for some symbols defined in the same linker output file
  - Permits overriding malloc() and free() in libc via LD_PRELOAD as an example
  - Symbols accessed via GOT indirection are **preemptible** symbols

Preemptible Symbols (2)

- ELF symbols have a visibility property in addition to binding (local vs global)
  - “default” global symbols are exported to other output files and are preemptible at run time
  - “hidden” global symbols are not exported to other output files and are not preemptible
    - Think of these as local symbols shared by multiple object files within a library or executable
  - “protected” global symbols are exported to other output files, but are not preemptible within the containing output file

Calling Preemptible Functions in PIC

- The code generation for PIC could read function pointers from the GOT and branch to them as indirect function calls similar to accessing data variables via GOT indirection

- But ELF does not do that…
  - Some libraries have many “remote” function calls, but individual processes may only invoke a subset. However, processing all the dynamic relocations for those GOT entries would add overhead in the run-time loader initialization.
  - PDE dynamic executables invoke “remote” functions, too

PLTs

- Static linker generates a **Procedure Linkage Table** (**PLT**) for calls to remote functions
- Each preemptible function is associated with a stub function (or “thunk”) in the PLT
- Each stub function reads a pointer from a separate PLT GOT (distinct from the normal GOT and also generated by the static linker) and branches to that pointer
- Each entry in the PLT is associated with a dynamic relocation (typically R_<arch>_JUMP_SLOT) whose symbol is the name of the remote function and the target address is the PLT GOT entry

Lazy PLT Resolution (1)

- To avoid the overhead of processing jump slot relocations on startup, PLT GOT entries are resolved on demand

- The first entry in the PLT is a special thunk, **PLT0**, which reads a function pointer and an opaque data pointer from two entries at the start of the PLT GOT and branches to the function pointer passing the opaque data pointer in a specific register

- PLT GOT entries are initialized so that on the first call, the PLT thunk stores the index of the PLT entry in a specific register and branches to the special PLT0 thunk

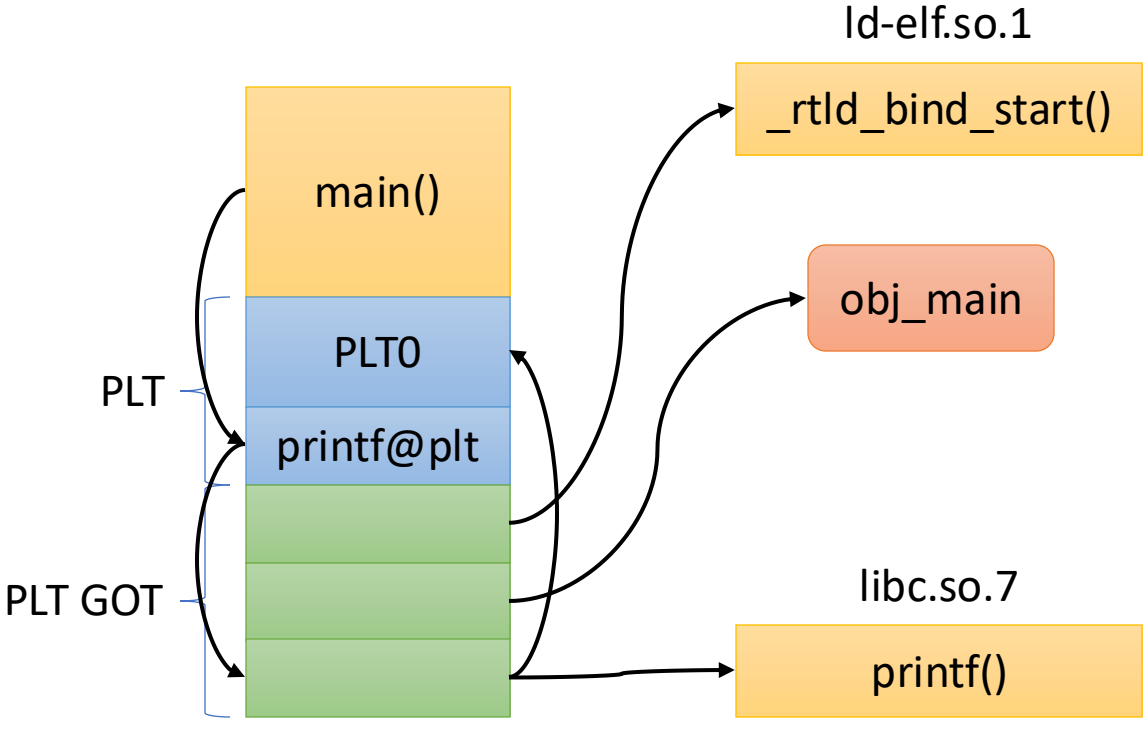
Lazy PLT Resolution (2)

- The run-time loader initializes the two entries at the start of the PLT GOT with code and data pointers
  - For FreeBSD, the code pointer points to _rtld_bind_start() and the data pointer points to the associated Obj_Entry
- When the run-time loader is called, it uses the PLT index to locate the jump slot relocation and applies that relocation to update the PLT GOT entry
- The resolved function pointer address is also branched to directly at the end of the resolver routine

Lazy PLT Example

```
#include <stdio.h>

int main(void)
{
  printf("hello ");
  printf("world\n");
  return 0;
}
```



PDE Data and Code Access (Static Binary / Library)

```
int bar(int *);

extern int x;

int
foo(void)
{
    return bar(&x) + 4;
}
```

x86-64 gcc 14.2 -O2 -fno-pic

```
foo:
        subq    $8, %rsp
        movl    $x, %edi
        call    bar
        addq    $8, %rsp
        addl    $4, %eax
        ret
```

Absolute 32-bit Address of “x”

32-bit PC-relative Branch

PIC Data and Code Access (Shared Library)

```
int bar(int *);

extern int x;

int
foo(void)
{
    return bar(&x) + 4;
}
```

x86-64 gcc [illegible] -fPIC

```
foo:
        subq    $8, %rsp
        movq    x@GOTPCREL(%rip), %rdi
        call    bar@PLT
        addq    $8, %rsp
        addl    $4, %[illegible]x
        ret
```

Reads “&x” from GOT via PC-relative Address

PC-relative Branch to PLT Stub

PDE Data and Code Access (Dynamic Binary)

```
int bar(int *);

extern int x;

int
foo(void)
{
    return bar(&x) + 4;
}
```

x86-64 gcc 14.2 -O2 -fno-pic

```
foo:
        subq    $8, %rsp
        movl    $x, %edi
        call    bar
        addq    $8, %rsp
        addl    $4, %eax
        ret
```

What if “x” is in a shared library?

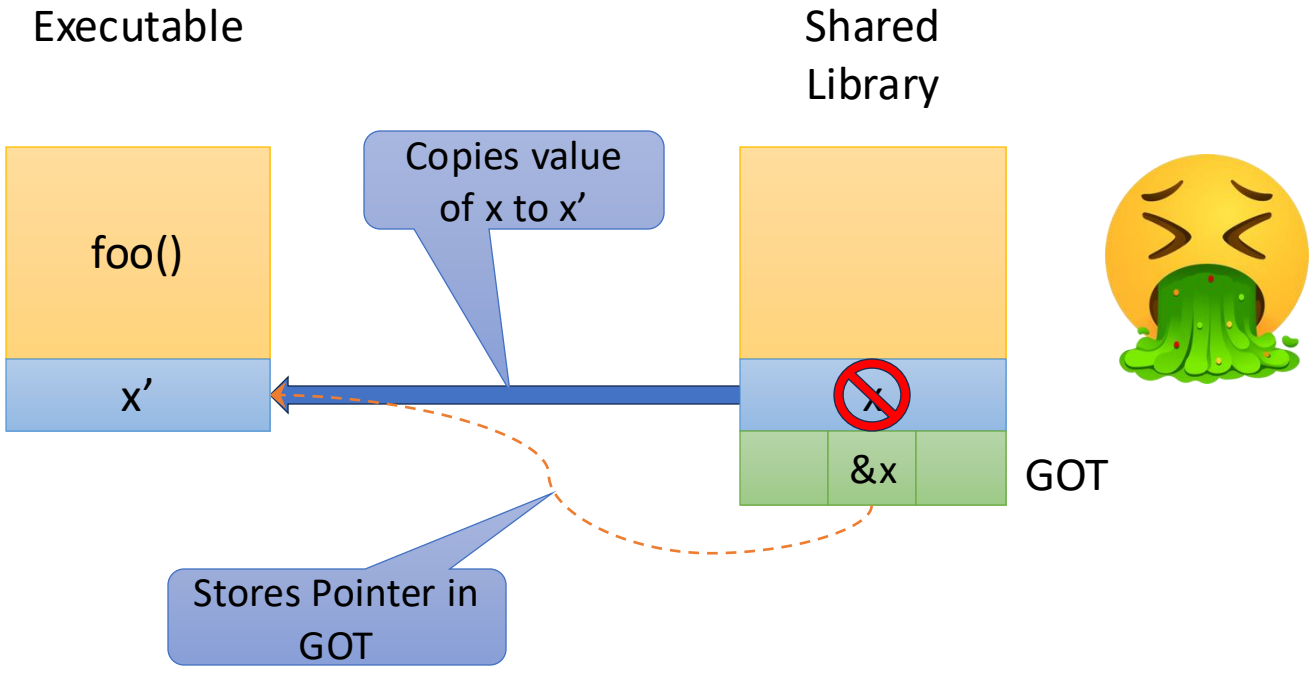
What if “bar” is in a shared library?

PDE Dynamic Executables

- Q: How to handle branches to external functions in shared libraries?
- A: PLT Stub
  - Linker rewrites target of branch to address of PLT stub

- Q: How to handle absolute addresses for external data in shared libraries?
- A: Copy Relocations
  - Static linker reserves space in .bss of executable for a copy of *x* (*x'*)
  - run-time loader copies value of *x* from shared library into *x'*
  - run-time loader resolves symbol lookups for *x* to *x'* instead

Copy Relocations



PDE Function Pointer (Static Binary / Library)

```c
#include <stdio.h>

void foo(void)
{
    printf("fclose = %p\n", &fclose);
}
```

x86-64 gcc 14.2 -O2 -fno-pic

```
.LC0:
        .string "fclose = %p\n"
foo:
        movl    $fclose, %esi
        movl    $.LC0, %edi
        xorl    %eax, %eax
        jmp     printf
```

Absolute Addresses

32-bit PC-relative Branch

PIC Function Pointer (Shared Library)

```c
#include <stdio.h>

void foo(void)
{
    printf("fclose = %p\n", &fclose);
}
```

x86-64 gcc 14.2 -

```
.LC0:
    .string "fclose = %p\n"
foo:
    movq    fclose@GOTPCREL(%rip), %rsi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    jmp     printf@PLT
```

Reads “&fclose” from GOT via PC-relative Address

PC-relative Branch to PLT Stub

PDE Function Pointer (Dynamic Binary)

```c
#include <stdio.h>

void foo(void)
{
    printf("fclose = %p\n", &fclose);
}
```

x86-64 gcc 14.2 -O2 [illegible]

```
.LC0:
        .string "fclo    = %p\n"
foo:
        movl    $fclose, %esi
        movl    $.LC0, %edi
        xorl    %eax, %eax
        jmp     printf
```

What absolute address to use for “&fclose”?

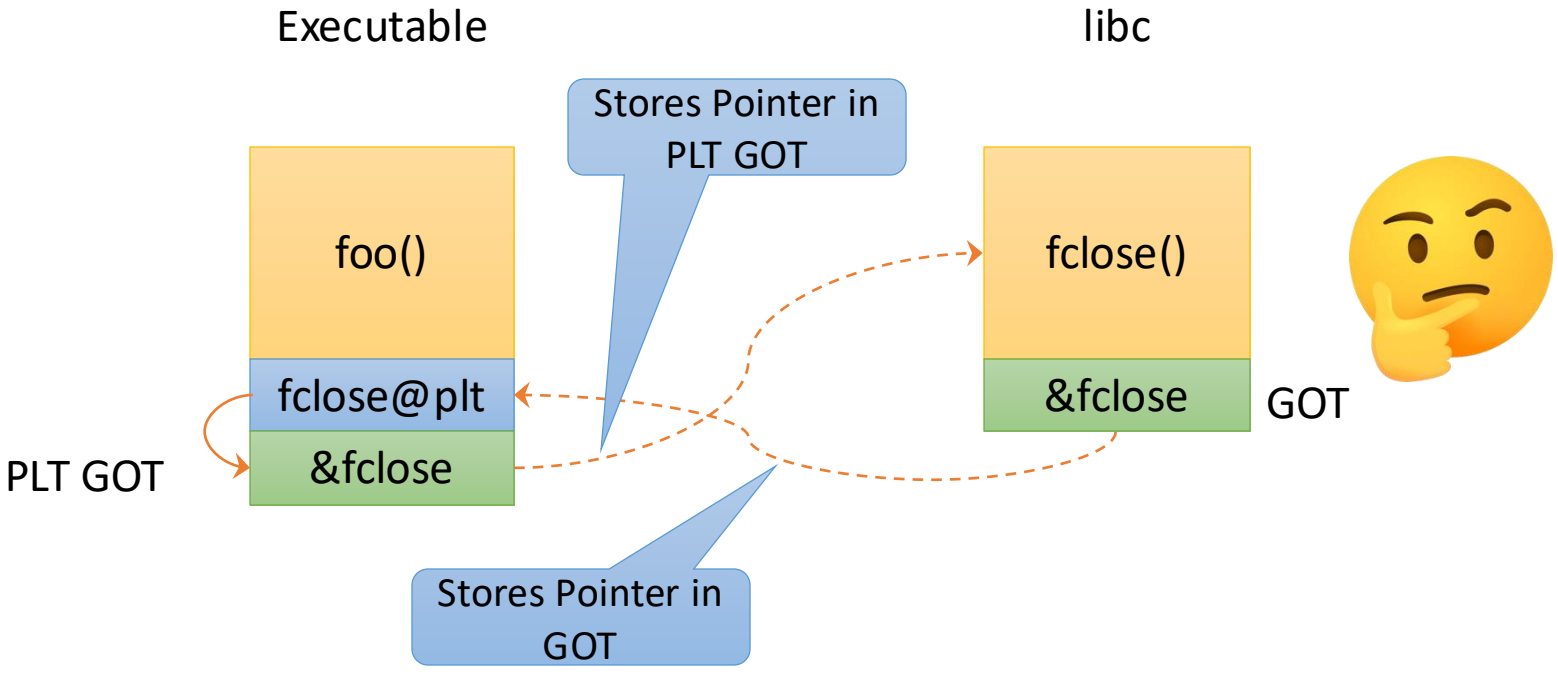
Canonical PLT Entries

- Q: What absolute address (fixed at link time) can we use for an external function pointer?
- A: Address of the PLT stub for that function in the executable.

- Q: How to handle C's requirement that all function pointers to the same function compare equal?
- A: All other data pointers to the function have to use the address of the PLT stub in the executable.

- Copy Relocations for Function Pointers

Canonical PLT Entries



What about PIE?

- -fPIE uses GOT indirection for data variables on clang (but not GCC!)
  - No copy relocations for clang
- -fPIE uses GOT indirection for function pointers on both GCC and clang
  - No canonical PLT entries

PIE Data and Code Access (Dynamic Binary) - GCC

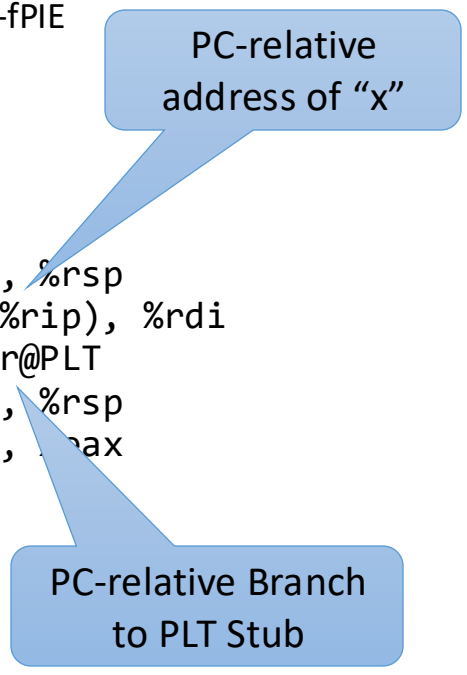
```
int bar(int *);

extern int x;

int
foo(void)
{
      return bar(&x) + 4;
}
```

x86-64 gcc 14.2 -O2 -fPIE

```
foo:
      subq    $8, %rsp
      leaq    x(%rip), %rdi
      call    bar@PLT
      addq    $8, %rsp
      addl    $4, %eax
      ret
```



PIE Data and Code Access (Dynamic Binary) - clang

```
int bar(int *);

extern int x;

int
foo(void)
{
      return bar(&x) + 4;
}
```

x86-64 clang 19.1.0 -O2 -fPIE

Identical to -fPIC

```
foo:
      pushq   %rax
      movq    x@GOTPCREL(%rip), %rdi
      callq   bar@PLT
      addl    $4, %eax
      popq    %rcx
      retq
```

PIE Function Pointer (Dynamic Binary)

```
#include <stdio.h>

void foo(void)
{
    printf("fclose = %p\n", &fclose);
}
```

x86-64 gcc 14.2 -O2 –fPIE

Identical to -fPIC

```
.LC0:
    .string "fclose = %p\n"
foo:
    movq    fclose@GOTPCREL(%rip), %rsi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    jmp     printf@PLT
```

Indirect Functions (IFUNC)

- Indirect functions permit resolving a symbol to the address of another symbol at run time
  - Typically used to provide optimized versions of functions (e.g. SSE2 vs AVX)

- The value of symbols of type STT_GNU_IFUNC is the address of a **resolver function**
  - The resolver function returns the resolved symbol value

- When resolving the address of an indirect function symbol, the run-time loader calls the resolver function to obtain the final symbol value

- R_<arch>_IRELATIVE relocations are a relative relocation whose initial value is also the address of a resolver function

Miscellaneous Notes

- Default code generation varies by compiler and architecture
- -fno-plt is a recent innovation to disable PLT indirection
  - Function calls read from the GOT directly
- Some toolchain folks are advocating for GOT indirection for PDE (-fno-pic) to eliminate copy relocations and canonical PLTs entirely
  - Linker relaxations may be able to relax GOT indirection back to PDE-like direct access at static link time
- Static libraries can contain PIC object files
  - Useful as an input when linking a shared library

Guide to Relevant ELF File Sections

Name	Object Files	Linker Output Files	Description
.text	✓	✓	Machine code
.data	✓	✓	Writable global variables
.rodata	✓	✓	Read-only global variables, constant pools
.bss	✓	✓	Writable global variables initialized to 0
.rel[a].<section>	✓		Static ELF relocations
.got		✓	Data GOT
.plt		✓	PLT stubs
.got.plt		✓	PLT GOT
.rel[a].dyn		✓	Dynamic ELF relocations for everything but PLT GOT
.rel[a].plt		✓	Dynamic ELF relocations for PLT GOT

MIPS

More reasons I dislike MIPS

- MIPS is a special snowflake among ELF architectures
  - I think it suffered perhaps from being “first”
- No dynamic ELF relocations for the GOT, instead the single GOT is split into separate regions with implicit rules for resolving GOT entries for each region
  - “Local” region are all relative relocations without a symbol (DT_MIPS_LOCAL_GOTNO entries)
  - “Global” region contains symbol addresses where the GOT entries are associated with a contiguous range of indices in the symbol table (starting at DT_MIPS_GOTSYM symbol index)
  - PLT stubs use the end of the “Global” region (starting at DT_MIPS_SYMTABNO)
- Index passed to run-time loader’s resolver from PLT stubs is an index into the symbol table
  - GOT index is computed as DT_MIPS_LOCAL_GOTNO + (symbol index – DT_MIPS_GOTSYM)

Resources

- Linkers and Loaders by John Levine
  - Only book I'm aware of that talks about this topic…
  - … but it was published in 1999 while was an undergrad

- Compiler Explorer: https://godbolt.org

- MaskRay (LLVM developer)'s blog: https://maskray.me/blog/