ELF1 7B ELF Sections for Relocation - ELF Study 1999

Young W. Lim

2020-05-11 Mon

Outline

1. Based on

2. ELF sections for relocation
   - TOC
   - sections for global and static variables
   - .bss and .data sections
   - sections for relocations
   - .dynamic section

Based on

"Study of ELF loading and relocs", 1999
http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

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

TOC: ELF special sections

- Sections for global and static variables
- `.bss` and `.data` sections
- Sections for relocations
- `.dynamic` section

TOC: Sections for global and static variables

- static and automatic variables
  - initialization
  - storage locations
  - default uninitialized value
- universal zero initializer
- initialized and uninitialized static variables

static and automatic variables - initialization

- static variables
  - *initialized* at compile time,
    since their address is known and fixed.
  - initialization to zero does not incur a run time cost
- automatic variables
  - *initialized* at run time
  - incurs run time cost
    each time the function is called
  - different addresses for each different call
  - if you do need that initialization, then request it.

`https://stackoverflow.com/questions/14049777/why-are-global-variables-always-init`

static and automatic variables - storage locations

- static variables are stored (either global or local)
  - in the .data section when initialized
  - in the .bss section when uninitialized
  - a fixed memory location is allocated at compile time.
  - thus, have '0' as their default values.
- auto variables are stored
  - on the stack, not a fixed memory location
  - stack memory is allocated at run time
  - thus, have their default value as *garbage*

https://stackoverflow.com/questions/14049777/why-are-global-variables-always-init

static and automatic variables - default uninitialized value

- an object with automatic storage duration
  uninitialized value is indeterminate
- an object that has static storage duration
  defautt uninitialized values:
  - if it has pointer type, a null pointer;
  - if it has arithmetic type, (signed or unsigned) zero;
  - apply the above two rules to the belows
    - if it is an aggregate, every member is initialized
    - if it is a union, the first named member is initialized

`https://stackoverflow.com/questions/13251083/the-initialization-of-static-variable`

universal zero initializer

- the universal zero initializer
  - initializes everything in an object to 0, whether it's static or not
  -
    ```
    sometype      var = {0};
    someothertype var[SOMESIZE] = {0};
    anytype       var[SIZE1][SIZE2][SIZE3] = {0};
    ```

https://stackoverflow.com/questions/13251083/the-initialization-of-static-variable

initialized and uninitialized static variables

- global static / local static variables / arrays
  - initialized static variables
    - given value from code at compile time
    - *usually* stored in `.data`
      though this is compiler specific
  - uninitialized static variables
    - initialized at run time
    - stored into `.bss`
      though again this is compiler specific

https://stackoverflow.com/questions/13251083/the-initialization-of-static-variable

TOC: .bss and .data sections (1)

- .bss

  (1) to be intialized to zero

  (2) no zeros in the file

  (3) PROGBITS vs NOBITS

  (4) unintilialized global - COMMON

  (5) global static and local static variables

  (6) -fno-common

  (7) -fno-common error messages

- .data

TOC: .bss and .data sections (2)

- `.rodata`
- `.data.rel.ro`
  (1) after relocation
  (2) relo
  (3) initialized global variables
  (4) dynamic relocation

.bss (1) to be initialized to zero

- The .bss section is guaranteed to be all zeros when the program is loaded into memory.
- the .bss section can have global data
  - uninitialized
  - initialized to zero
- `static int g_myGlobal = 0;      // <--- in .bss section`

https://stackoverflow.com/questions/16557677/difference-between-data-section-and-t

.bss (2) no zeros in the file

- the .bss section data are not included in the ELF file on disk
  - there isn't a whole region of zeros in the file for the .bss section
- instead, the loader knows from the section headers how much to allocate for the .bss section, and simply zero it out before transfer control

https://stackoverflow.com/questions/16557677/difference-between-data-section-and-t

.bss (3) PROGBITS vs NOBITS

- the readelf -S section headers output:

```
[ 3] .data PROGBITS 00000000 000110 000000 00 WA 0 0 4
[ 4] .bss  NOBITS   00000000 000110 000000 00 WA 0 0 4
```

- .data is marked as PROGBITS
  - there are "bits" of program data in the ELF file
    that the loader needs to read out into memory
- .bss is marked NOBITS
  - there's nothing in the file
    that needs to be read into memory as part of the load

https://stackoverflow.com/questions/16557677/difference-between-data-section-and-t

.bss (4) uninitialized global - COMMON

- uninitialized global data (block started by symbol)
- depending on the compilers, uninitialized global variables could be stored in a nameness section called COMMON (named after Fortran 77's "common blocks")

```
int globalVar;
static int globalStaticVar;
void dummy() {
   static int localStaticVar;
}
```

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.bss (5) global static and local static variables

- compile with gcc -c, then on x86_64,
  the resulting object file has the following structure:
- only the uninitialized file-scope static variables
  and uninitialized local-scope static variables
  (globalStaticVar or localStaticVar)
  are in the .bss section
- uninitialized file-scope global variables in COMMON

```
$ objdump -t foo.o

SYMBOL TABLE:
 ....
0000000000000000 l     O .bss   0000000000000004 globalStaticVar
0000000000000004 l     O .bss   0000000000000004 localStaticVar.1619
 ....
0000000000000004       O *COM*  0000000000000004 globalVar
```

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.bss (6) -fno-common

- If one wants globalVar to reside in the .bss section, use the -fno-common (encouraged)
- compile with gcc -c, then on x86_64, no error / no warning messages without -fno-common

```
$ cat foo.c
int globalVar;                  // int
$ cat bar.c
double globalVar;               // double
int main(){}
$ gcc foo.c bar.c
```

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.bss (7) -fno-common error messages

- there is no error message about redefinition of the same symbol in both source files (notice we did not use the `extern` keyword here),
- there is no complaint about their different data types and sizes either.
- However, if one uses `-fno-common`, the compiler will complain:

```
/tmp/ccM71JR7.o:(.bss+0x0): multiple definition of ‘globalVar’
/tmp/ccIbS5MO.o:(.bss+0x0): first defined here
ld: Warning: size of symbol ‘globalVar’ changed from 8 in /tmp/ccIbS5MO.o to 4
```

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.data

- Initialized data.

`https://www.cs.stevens.edu/~jschauma/631A/elf.html`

.rodata

- Read-only data.

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.data.rel.ro (1) after relocation

- Similar to .data section, but this section should be made Read-Only after relocation is done.

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.data.rel.ro (2) relro

- gcc (the GNU linker), and glibc (the dynamic linker) cooperate to implement read-only relocations, or relro.
- a part of an executable or shared library is designated as being read-only after dynamic relocations have been applied.

https://stackoverflow.com/questions/7029734/what-is-the-data-rel-ro-used-for

.data.rel.ro (3) initialized gloabl variables

- used for read-only global variables
  which are initialized by
    - the address of a function or
    - a different global variable
    - these themselves also require relocations

https://stackoverflow.com/questions/7029734/what-is-the-data-rel-ro-used-for

.data.rel.ro (4) dynamic relocation

- Because such an initialized global variable requires a runtime initialization in the form of a dynamic relocation, it can not be simply placed in a read-only segment.
- it will be declared as constants (read-only) after the initialization (dynamic relocation), and will not be changed by the program
- therefore the dynamic linker can mark it as read-only after the dynamic relocation has been applied.

`https://stackoverflow.com/questions/7029734/what-is-the-data-rel-ro-used-for`

TOC: sections for relocations (1)

- sectons in relocatable object files
- relocation table sections
- multiple relocation sections but a single table
- `.rel.XXX`, `.rela.XXX`
- `.rela.text`, `.rel.text`
- `.rel.text` and `.rel.data` sections
- `.rel.text` section
- `.rel.data` section

TOC: sections for relocations (2)

- .rel.dyn
- rela.dyn
- rela.plt
- .got
- .got.plt
- .plt

sections in relocatable object files (1)

.text	- the machine code of the compiled program.
.rodata	- read-only data, such as the format strings in printf statements.
.data	- *initialized global* variables.
.bss	- *uninitialized global* variables. - BSS stands for Block Storage Start - occupies no space in the object file merely a placer holder.
.symtab	- a symbol table with information about functions and global variables defined and referenced in the program. - no entries for local variables which are maintained on the stack.

https://www.linuxjournal.com/article/6463

sections in relocatable object files (2)

`.rel.text`	- a list of *locations* in the `.text` section that need to be modified when the linker combines this object file with other object
`.rel.data`	- relocation information for global variables *referenced* but *not defined* in the current module.
`.debug`	- a debugging symbol table with entries for local and global variables. - present only if compiled with -g
`.line`	- a mapping between line numbers in the C source and machine instructions in the `.text` - required by debugger programs.
`.strtab`	-a string table for the symbol tables #ERROR

`https://www.linuxjournal.com/article/6463`

relocation table sections (1)

old	current	listed relocs
.rel.bss	.rel.dyn	contains all the R_386_COPY relocs
.rel.data	.rel.dyn	contains all the R_386_32 and R_386_RELATIVE relocs
.rel.got	.rel.dyn	contains all the R_386_GLOB_DAT relocs
.rel.plt	.rel.plt	contains all the R_386_JUMP_SLOT relocs

- R_386_JUMP_SLOT relocs modify the first half of the GOT elements
- R_386_GLOB_DAT relocs modify the second half of the GOT elements

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

relocation table sections (2)

`rel.text` `rela.text`	compile time / static relocation table
`rel.dyn` `rela.dyn`	run time / dynamic relocation table
`rel.plt` `rela.plt`	run time / dynamic relocation table

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

multiple relocation sections but a single table

- an object file can have multiple relocation sections when creating the relocation table for an executable or shared object file,
- the link-editor catenates those sections to form a single relocation table.

- Although the sections may remain independent in the object file, the runtime linker sees a single table.
- When the runtime linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions.

https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS31DTE/DOCU_002.HTM#

.rel.XXX, .rela.XXX

- Compile-time/Static relocation table for other sections.
- For example, .rela.init_array is the relocation table for .init_array section.
- Whether to use .rel or .rela is platform-dependent.
  - for x86_32, .rel
  - for x86_64, .rela

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.rel.text, .rela.text

1. For programs compiled with -c option,
   this section provides information to the link editor ld
   where and how to patch executable code in .text section.

1. The difference between .rel.text and .rela.text
   - entries in .rel.text does not have addend member
   - instead, the addend is taken from the memory location described by offset member.
   - compare struct Elf64_Rel with struct Elf64_Rela in /usr/include/elf.h

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.rel.text and .rel.data sections

- .rel.text : relocation information for .text section
  - a list of locations in the .text section
    that will need to be modified
    when the linker *combines* this object file with others
- .rel.data : relocation information for .data section
  - a list of locations in the .data section
    that will need to be modified
    when the linker *combines* this object file with others

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

.rel.text section

- relocation information for .text section
- modify any instruction in the code section that
  - calls an external function
  - references a global variable
- do not modify any instructions in the code section that
  - calls local functions
- executable files do not include relocation information unless the user explicitly instructs the linker

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

.rel.data section

- relocation information for .data section
- relocation information for any global variable that are referenced or defined by the data section
- modify the intialized values of any global variable when the initialized values are
  - the address of a global variable (&cPub)
  - externally defined function (&fPub)

```
typedef struct { char* p; char (*f)(int); } _st;
_st a[] = { {&cLocal, fLocal}, {&cPub, fPub} }
```

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

.rel.dyn (1) relocation after loading

- The dynamic relocation section describes all locations within the object that must be *adjusted* if the object is loaded at an address other than its linked base address.
- Only one dynamic relocation section is used to resolve addresses in data items, and it must be called .rel.dyn

https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS31DTE/DOCU_002.HTM

`.rel.dyn` (2) normal relocation secton

- shared (dynamic) executable files can contain normal relocation sections (`.rel.text`) in addition to a dynamic relocation section (`.rel.dyn`)

- the normal relocation sections (`.rel.text`) may contain resolutions for any absolute values in the `main` program.

- the dynamic linker does not resolve these or relocate the `main` program.

`https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS31DTE/DOCU_002.HTM#`

.rel.dyn (3) an array of elements

- The dynamic relocation section is
  an array of entries of the following type Elf32_Rel

```
typedef struct {
    Elf32_Addr      r_offset;
    Elf32_Word      r_info;
} Elf32_Rel;
```

  - r_offset Identifies
    the location within the object to be adjusted.
  - r_info Identifies
    the relocation type and the index
    of the symbol that is referenced.
  - The macros ELF32_R_SYM and ELF32_R_TYPE
    access the individual attributes.

https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS31DTE/DOCU_002.HTM#

.rel.dyn (4) attributes of relocation table

- The entries of the dynamic relocation section are ordered by symbol index value.
- The DT_REL and DT_RELSZ entries
  of the .dynamic section
  describe the attributes of
  the dynamic relocation section
  the relocation table rel.dyn
  - DT_REL : the address of a relocation table
  - DT_RELSZ : the size of a relocation table

https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS31DTE/DOCU_002.HTM

.rela.dyn

- For dynamic binaries, .rela.dyn relocation table holds information of variables which must be relocated upon loading
- Each entry in this table is a struct Elf64_Rela (see /usr/include/elf.h) which has only three members:
  - offset : the variable's virtual memory address which holds the "patched" value during the relocation process [usually position-independent]
  - info : index into .dynsym section and relocation type
  - addend

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.rela.plt

- .rela.plt relocation table is similar to the one in .rela.dyn section;
    - .rela.plt is for functions
    - .rela.dyn is for variables
- The relocation type of entries in this table is R_386_JMP_SLOT or R_X86_64_JUMP_SLOT and the offset refers to memory addresses which are inside .got.plt section.
- .rela.plt table holds information to relocate entries in .got.plt section.

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.got

- For dynamic binaries,
  this Global Offset Table holds the addresses of variables
  which are relocated upon loading.

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.got.plt

- For dynamic binaries,
  this Global Offset Table holds
  the addresses of functions in dynamic libraries.
- They are used by trampoline code in .plt section.
- If .got.plt section is present,
  it contains *at least three* entries,
  which have special meanings.

https://www.cs.stevens.edu/~jschauma/631A/elf.html

.plt

- For dynamic binaries,
  this Procedure Linkage Table holds
  the trampoline/linkage code.

https://www.cs.stevens.edu/~jschauma/631A/elf.html

TOC: .dynamic section

- .dynamic section
- .dynamic section - an array of the dynamic structures
- Program header table element of the type PT_DYNAMIC
- Array structure of the .dynamic section
- .dynamic section - runtime linker
- .dynamic section - dynamic linker's behavior
- Loading the necessary shared libraries
- link_map structure
- Linking external functions
- Searching Link_map structure

.dynamic section

- the dynamic linker uses .dynamic section to bind procedure addresses such as the symbol table and relocation information

Computer Architecture : A Programmer's Perspective

.dynamic section - an array of the dynamic structures

- if an object file participates in dynamic linking, its program header table will have an element of type PT_DYNAMIC.
- this segment contains the .dynamic section.
  - is labeled by a special symbol, _DYNAMIC
  - contains an array of the dynamic structures

https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-42444.html

Program header table element of the type PT_DYNAMIC

- If an bject file participates in dynamic linking, its program header table will have an element of type PT_DYNAMIC
- program header table structure

```
typedef struct {
      Elf32_Word      p_type;    // the kind of segment ... PT_DYNAMIC
      Elf32_Off       p_offset;  // from the beginning of the file
      Elf32_Addr      p_vaddr;   // virtual address
      Elf32_Addr      p_paddr;   // physical address
      Elf32_Word      p_filesz;  // size of the file image of the segment
      Elf32_Word      p_memsz;   // size of the memory image of the segment
      Elf32_Word      p_flags;
      Elf32_Word      p_align;
} Elf32_Phdr;
```

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblk/index.html#chapter6-42444

Array structure of the .dynamic section

- the segment whose program header table type is PT_DYNAMIC contains the .dynamic section
    - has the label _DYNAMIC
    - contains an array of the following structure

        -
          ```
          typedef struct {
            Elf32_Sword d_tag;
              union {
                Elf32_Word   d_val;
                Elf32_Addr   d_ptr;
                Elf32_Off    d_off;
              } d_un;
          } Elf32_Dyn;
          ```
        - d_tag controls the interpretation of d_un

https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblk/index.html#chapter6-42444

DT_RELA and DT_REL

- the address of a relocation table (rel.dyn)
- This element requires
  the DT_RELASZ and DT_RELAENT elements also be present.
- When relocation is mandatory for a file,
  either DT_RELA or DT_REL can occur.

https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS31DTE/DOCU_002.HTM#

DT_RELASZ and DT_RELEAENT

- DT_RELASZ
  Contains the size in bytes of the DT_RELA relocation table.
  (Not used by the default system linker and loader.)
- DT_RELAENT
  Contains the size in bytes of a DT_RELA relocation table entry.
  (Not used by the default system linker and loader.)

https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS31DTE/DOCU_002.HTM

.dynamic section - runtime linker

- the runtime linker (dynamic linker), can locate its own dynamic structure through _DYNAMIC symbol, even when relocation entries have not yet been processed
- the runtime linker must initialize itself without relying on *other programs* to relocate its memory image.

https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-74186.html

.dynamic section - dynamic linkers's behavior

- .dynamic section essentially holds
  a number of arguments
  that inform on and influence
  parts of the dynamic linker's behavior

- as a component of the run-time, the dynamic linker does
  many other things besides just *relocate functions*,
  it also executes other house keeping functions
  like INIT and FINI

- see elf/elf.h

http://blog.k3170makan.com/2018/11/introduction-to-elf-format-part-vii.html

Loading the necessary shared libraries

- When the dynamic linker is mapped to the memory,
  it first handles its own relocations.
- Then, it looks into the `.dynamic` section and
  searches for `DT_NEEDED` tags
  to locate the different shared libraries to be loaded.
- It then brings the shared library in memory,
  looks into its `.dynamic` section and
  adds the library's symbol table to a chain of symbol tables it maintains.
- It also creates an `link_map` entry for every shared library
- the first entry in `link_map` is of the executable binary itself.

https://gist.github.com/DhavalKapil/2243db1b732b211d0c16fd5d9140ab0b

link_map structure

-
  ```
  struct link_map
  {
    ElfW(Addr)         l_addr;
    char *             l_name;
    ElfW(Dyn) *        l_ld;
    struct link_map *  l_next, *l_prev;
    ElfW(Dyn) *        l_info[DT_NUM + DT_THISPROCNUM + DT_VERSIONTAGNUM +
                              DT_EXTRANUM + DT_VALNUM + DT_ADDRNUM];
  };
  ```
- l_addr base address shared object is loaded at
- l_name absolute file name object was found in
- l_ld dynamic section of the shared object
- l_next, l_prev chain of loaded objects
- l_info[...] holds pointers to symbol table (l_info[DT_SYMTAB]) and relocation table (l_info[DT_JMPREL])

https://gist.github.com/DhavalKapil/2243db1b732b211d0c16fd5d9140ab0b

Linking external functions

- During the process of linking external functions,
  a call is made to _dl_runtime_resolve with parameters:
  - the link_map struct and
  - the index into the relocation table for that function.
- The relocation entry gives
  - the index in the symbol table for that function and also
  - the address in GOT to be patched.
- The symbol is then searched in shared libraries
  using the link_map struct.

https://gist.github.com/DhavalKapil/2243db1b732b211d0c16fd5d9140ab0b

Searching `link_map` structure

- The search involves the following steps:
  - Generating a hash of the symbol name to be searched for.
  - Lookup the symbol table entry using that index.
  - Lookup the name of that symbol in string table and compare.
- If found, the symbol's address is added
  to the corresponding shared library's base address

`https://gist.github.com/DhavalKapil/2243db1b732b211d0c16fd5d9140ab0b`