1 Linking[1]

1.1 Basics

1. C compiler driver:

   cpp C preprocessor. *.c → *.i

   cc1 C compiler. *.i → *.s

   as assembler. *.s → *.o

   ld linker. *.o → exec

2. Task of linker:

   Symbol resolution Associate each symbol reference with one symbol definition.

   Relocation Associate a memory location with each symbol definition, and modify all references to the symbol so that they point to this memory location.

3. Object files:

   relocatable object file Binary code + data.

   executable object file Can be directly copied into memory and executed.

   shared object file A special type of relocatable object file. Can be loaded into memory and linked dynamically, at either load time or run time.

1.2 Relocatable object file

ELF header 16-byte sequence. Word size & byte ordering of the system. Information for linker to parse & interpreting the object file: size of ELF header; object file type; machine type; file offset, size, number of entries of the section header table.

section header table a fixed-sized entry for each section. Locations & sizes of the sections.

.text Machine code of the compiled program.

.rodata Read-only data.

.data Initialized non-zero global / static variables.

.bss Merely place holder (does not occupy actual space). Uninitialized or zero-initialized global / static variables.

symtab Symbol table.

.rel.text Relocation information. A list of locations in `.text` that need to be modified when the linker combines this object file with others. Any instruction that calls an external function or references a global variable needs to be modified.

[1] 7.8-7.14 omitted for the moment.

.rel.data Relocation information. Initialized global variable whose initial value is the address of a global variable or an externally defined function.

.debug A debugging symbol table only present with `-g` option.

.line A mapping between line numbers in the original C code and machine code instructions in `.text`.

.strtab A string table for the symbol tables in `.symtab` and `.debug`, and section names in the section headers.

1.3 Symbol table

global symbols Defined by this module and can be used by other modules. Non-static C functions & global variables.

external symbols C functions & global variables defined by other modules and used by this module.

local symbols C functions & variables defined and referenced exclusively by this module. `static` C functions & global variables.

Entry in symbol table:

```
typedef struct {
  int name;   /*String table offset. Name of the symbol.*/
  char type:4, /*Function or data (4 bits)*/
       binding:4; /*Local or global*/
  char reserved; /*Unused*/
  short section; /*Section header index*/
  long value; /*Address. Section offset (relocatable) or absolute address (
      executable)*/
  long size; /*Object size in bytes.*/
} Elf64_Symbol;
```

Pseudo sections:

ABS Should not be relocated

UNDEF Undefined symbols (referenced in this module but defined elsewhere).

COMMON Uninitialized data objects that are not yet allocated. In this case, `value` gives alignment requirement and `size` gives the minimum size. Difference from `.bss`:

- COMMON: uninitialized global variables.
- .bss: uninitialized static variables; global / static variables initialized to 0.

1.4 Symbol resolution

1.4.1 Multiply defined global variables

- Strong symbols: Functions; initialized variables.
- Weak symbols: Uninitialized variables.

Rules to resolve multiply defined global symbols:

- Multiple strong symbols are not allowed.
- Given a strong symbol and multiple weak symbols, choose the strong one.
- Given multiple weak symbols, choose any of them.

Error due to 2nd rule:

```
/*foo.c*/
#include <stdio.h>
void f(void);
int y = 15212;
int x = 15313;
int main() {
  f();
  printf("x=0x%x, y=0x%x\n", x, y);
  return 0;
}

/*bar.c*/
double x;
void f() { x = -0.0; }
```

Running the program will give `x=0x0, y=0x80000000`.

1.4.2 Static libraries

During the symbol resolution phase, the linker scans the relocatable object files and archives (static libraries) from left to right according to their order in the command line. During the scan, the following sets are maintained:

E Relocatable object files to be merged into the executable.

U Unresolved symbols.

D Symbols defined in previous input files.

- Initially **E, U, D** are all empty.
- For each input file *f*, if *f* is an object file, the linker adds *f* to **E** and updates **U, D** according to symbol definitions and references in *f*.
- If *f* is an archive, the linker attempts to match symbols in **U** against symbols defined by the members of the archive. If a referenced symbol in **U** is defined in a member *m* of the archive, then *m* is added to **E** and **U, D** are updated accordingly. Repeat until **U, D** no longer changes. Members of the archive not contained in **E** are discarded.
- If **U** remains nonempty after the scan, the linker prints an error and terminates. Otherwise it merges all relocatable object files in **E** to build the executable.

This makes the order of the command line arguments important.

- Archives are put at the end of the command line.

- If archives are dependent, they need to be ordered so that for each symbol s externally referenced by a member of an archive, at least one definition of s follows a reference to s on the command line. For this purpose archives can be repeated on the command line.

1.5 Relocation

Relocation includes two steps:

- Relocate sections & symbol definitions
  - Merges all sections of the same type into an aggregate section (e.g. `.data`).
  - Assigns run-time memory address to each **aggregate section**, each **(original) section** defined by input modules and each **symbol** defined by input modules.
- Relocate symbol references within sections
  - Modifies symbol references in `.text` and `.data` so that they point to the correct run-time addresses according to the **relocation entries** in `.rel.text` and `.rel.data`.

When the assembler encounters an reference to an object whose ultimate address is unknown, a **relocation entry** is generated for this reference.

```
typedef struct {
  long offset; /*section offset of the reference to relocate*/
  long type:32, /*Relocation type. */
       symbol:32; /*Symbol table index*/
  long addend; /*Constant part of relocation expressions*/
} Elf64_Rela;
```

There are totally 32 relocation types. Basic types that support **small code model**(data + code $<$ 2GB):

- R_X86_64_PC32: for references using 32-bit PC-relative address.
- R_X86_64_32: for references using 32-bit absolute address.

The pseudo code for the relocation algorithm used by the linker:

```
for section s {
  for relocation entry r {
    refptr = s + r.offset; /*ptr to reference to be relocated*/

    if(r.type == R_X86_64_PC32) {
      refaddr = ADDR(s) + r.offset;
      *refptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr);
    } else if (r.type == R_X86_64_32)
      *refptr = (unsigned)(ADDR(r.symbol) + r.addend);
  }
}
```