Notes for CSC 2/454, Dec. 2, 2020

==========

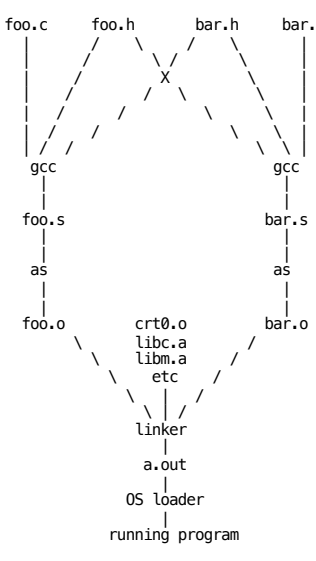
Building a program:

Terminology:
- An ***object file*** contains machine language code and data.
- A ***relocatable*** object file contains the information needed to relocate the file's contents.
- An ***executable*** object file can be loaded and run.

(You may recall that Rust, for some reason, calls object files ***crates***.)

It is possible for a file to be both relocatable and executable.

Example of a C program with 4 source files, `foo.c`, `foo.h`, `bar.c`, and `bar.h`:



Assemblers

Translate assembly language to machine language.
Long ago, had lots of fancy features (e.g., sophisticated macro systems) for the convenience of human users. Nowadays very little assembly code is written by hand.

Some produce assembly code and make the assembler a separate pass.
Some compilers produce machine code directly.

For these you either need an option to produce assembly code on demand or a good disassembler for people developing/debugging the compiler.

Principal complication of assembly is the fact that a label may be used before it is defined:

```
        cmp   %eax, %ecx
        jne   .L1
...
.L1:    addl  %eax, %edx
```

When the assembler sees `jne` (jump if not equal) the first time, it doesn't know `.L1`'s location.

Translation therefore takes 2 steps:

1) associate memory locations with labels, based on an understanding of how long each eventual code block will be (this can be complicated by the fact that the length of some instructions [e.g., branches, loads] depends on how far away things end up).

2) go back and do the actual assembly-to-machine code translation, using the locations figured out in step 1.

Step 2 also generates a symbol table.
Each entry contains
- the string representing the symbol
- the segment -- e.g. undefined, absolute, text, data, bss (bss = zero-initialized globals ["block started by symbol"])
- the offset from the start of the segment
- a bit for private versus global
- for symbols not defined here, a list of the instructions in which the symbol is referenced (so the linker can patch them up [see below])

This is in addition to (or an augmentation of) the symbol table produced by the compiler.

Linking

Assemblers (and compilers) seldom produce exactly the bits that will be in the code segment in memory when your program runs. Two tasks generally remain to be done

(1) ***Symbol resolution***

Most programs are made of separately-compiled modules.
Something needs to stitch these together to make a whole program. This is called ***linking***; it's done by a ***linker***.
The '.o' files that the assembler produces from your source files are called ***object*** files because they contain "object" code (as opposed to source code). They define certain ***symbols*** that represent interesting things in your program -- mainly code and data -- and contain ***unresolved references*** to symbols in other object files.
The linker takes a collection of object files and resolves mutual references. It usually knows about certain "standard" libraries that contain many of the symbols.

(2) ***Relocation***

Because your program is typically made from separately-compiled pieces, the assembler doesn't know when it creates a given .o file where in your address space that file will lie. This means it doesn't know the absolute addresses at which code and data will lie.
Branches can be made in terms of relative offsets from the program counter, but jumps, loads, and stores have to be deferred until we know what the absolute address of the beginning of the object file will be.
Once we know this address, we can ***relocate*** the code. This job is usually also done by the linker.
Object files contain information indicating that certain words need to be modified to reflect where symbols have been placed.
- Might be as simple as adding the address of a file to the word
- Or adding some piece of the address to some piece of the word; more on this below.

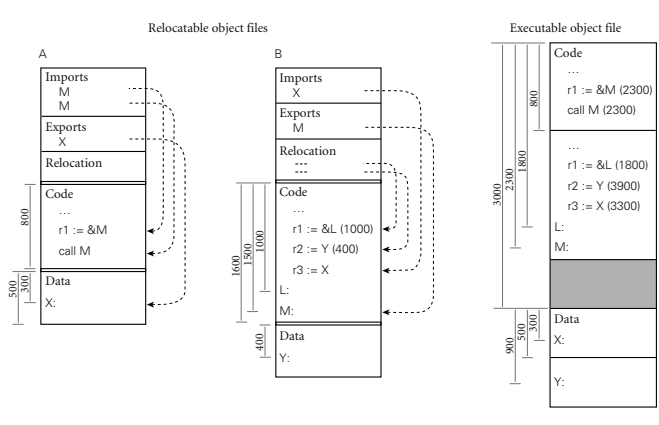


Figure 15.9 **Linking relocatable object files** A and B **to make an executable object file.** For simplicity of presentation, A's code section has been placed at offset 0, with B's code section immediately after, at offset 800 (addresses increase down the page). To allow the operating system to establish different protections for the code and data segments, A's data section has been placed at the next page boundary (offset 3000), with B's data section immediately after (offset 3500). External references to M and X have been set to use the appropriate addresses. Internal references to L and Y have been updated by adding in the starting addresses of B's code and data sections, respectively.

A warning: the term ***loading*** is sometimes used for relocation. It is better used for the task of putting a program (or at least part of it) into physical memory so it can run. The kernel does loading in response to an exec system call (or its equivalent in non-Unix systems). Once upon a time, when hardware didn't do address translation, programs had to be relocated when they were loaded; hence the confusion. It's especially unfortunate that Unix's linker is called "ld", which suggests "loader". Sometimes a linker is called a "link editor" or (unfortunately) "link-loader".

AND... Just to make life more confusing, modern systems often employ Address Space Layout Randomization (ASLR) as a security measure. This effectively puts relocation ***back*** into the loader's job description.

Unix ELF Object File Format (Executable and Linking Format)

Contains
- ELF header (contains pointer to section header table)
- sections

.text	code
.rodata	constants
.data	initialized, writable data
.bss	placeholder for uninitialized data
.symtab	global symbols, defined and undefined
.rel.text	relocation information for code
.rel.data	relocation information for data
.debug	debugger symbol table if compiled -g
.line	line number map if compiled -g
.strtab	heap for strings in .symtab and .debug

- section header table

ELF File Header

from /usr/include/sys/elf.h :

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf32_Half e_type;          /* Object file type */
  Elf32_Half e_machine;       /* Architecture */
  Elf32_Word    e_version;       /* Object file version */
  Elf32_Addr    e_entry;         /* Entry point virtual address */
  Elf32_Off     e_phoff;         /* Program header table file offset */
  Elf32_Off     e_shoff;         /* Section header table file offset */
  Elf32_Word    e_flags;      /* Processor-specific flags */
  Elf32_Half e_ehsize;        /* ELF header size in bytes */
  Elf32_Half e_phentsize;     /* Program header table entry size */
  Elf32_Half e_phnum;         /* Program header table entry count */
  Elf32_Half e_shentsize;     /* Section header table entry size */
  Elf32_Half e_shnum;         /* Section header table entry count */
  Elf32_Half e_shstrndx;      /* Section header string table index */
} Elf32_Ehdr;
```

Details of the relocation information vary from machine to machine.
ELF defines 11 different encodings.
Two of them cover most cases on the x86:
- PC relative branches
  linker should subtract address of instruction from target address and then add result into field (usually -4)
- absolute jumps
  linker should add target address into field (usually zero)

RISC machines tend to be quite a bit trickier.
For example, a source statement like

```
void() *f = &foo;
```

is likely to become a PAIR of instructions on even a 32-bit RISC machine:

```
lui r1, c1      # &foo >> 16
ori r1, c2      # &foo & 0xffff
```

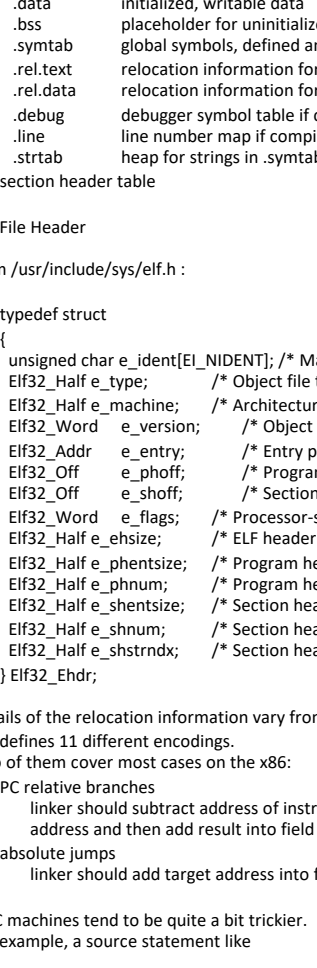
The linker needs to know how to create the two specified constants, given the address of foo, and how to embed them in the immediate fields of the instructions.

Loader: Loads file from disk/secondary storage

Read header for size of text and data segments
Create new address space - text, data, stack
Copy instructions/data from file into new address space (memory)
Copy program arguments onto stack
Initialize machine registers/stack pointer
Jump to startup routine

call any static initializers
copy program arguments from stack to registers (on RISC machine)
call program's main routine
on return, terminate program with exit system call

32-bit Linux Memory Layout (slightly updated from the version in the book).
Note: fig is not to scale -- kernel occupies 1/4 of address space.



Tools: Several tools can be used to read/interpret object files:

`od` -- displays the contents of any file
`nm` -- displays the symbol table information appended to an object file
`objdump` -- on Linux
`readelf` -- on Linux

(abbreviated) example

```
% nm -p -v time_test.o

time_test.o:
0000000000 f time_test.c
0000000000 U exit
0000000000 U random
0000000000 U printf
0000000004 D counter
0000000008 D nthreads
0000000044 d count
0000000048 d sense
0000002712 T main
0000003692 T barrier
0000003852 T initialize
0000136824 B t1
0000136828 B t2
0000136832 B t3
```

Key:

u	undefined (external)
t	text (code)
d	initialized data
b	bss
s	section boundary
f	source file boundary
a	absolute (non-relocatable) value

Capital letter means exported global.

Shared libraries

Motivation
- save disk space -- don't have copies of libraries in every executable on the disk
- save space in main memory -- don't have copies of libraries in every running process in memory
- allow upgrades of libraries without re-compilation -- when you replace the shared copy of the library you automatically upgrade every application that is set up to use it (at least the next time it is launched)

Implementation is kind of complicated. Key ideas include
- ***position-independent code*** (PIC)
- linkage tables (for absolute jumps, references to external symbols)
- initialization of tables with `ld.so` address, for lazy code linking

Lots of wrinkles may be different on different systems.
For example: x86-32 doesn't allow direct reads of PC (rip); need to fake with call instruction.

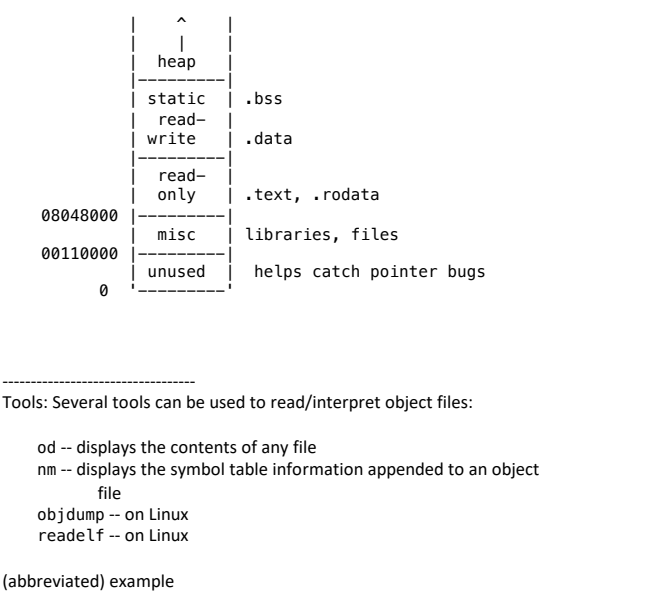


Figure 15.13 **A dynamically linked shared library.** Calls to `foo` and `bar` are made indirectly, using an address stored in the global offset tables (GOTs) of `main` and `foo`, respectively. Similarly, references to global variables `X` and `Y`, when made from `foo`, must employ a level of indirection. Resolved values are shown with dashed lines; initial values to support lazy linking (Section C-15.7.2) are shown with dotted lines. In the prologue of `foo`, register `ebx` is set to point to `foo`'s GOT, using `pc`-relative arithmetic.