Linking and Loading - Part II

Godmar Back

Virginia Tech

September 29, 2022

VIRGINIA TECH.
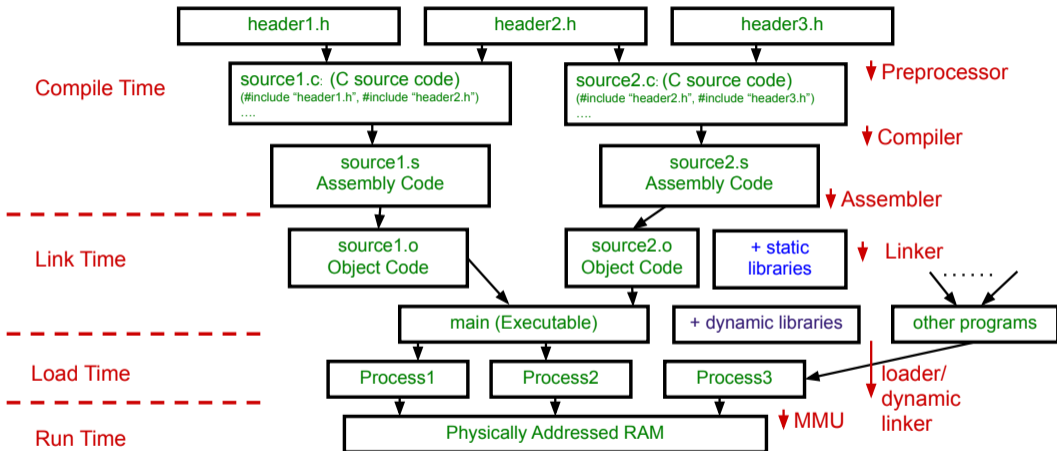
Software Engineering Aspects

Figure 1: Compilation, Linking, and Loading in a typical System

Local vs Global Symbols

source1.c

```
static int x;
static void f() {
    x = 1;
}

0000000000000000 t f
0000000000000000 b x
```

source2.c

```
static int x;
static void f() {
    x = 2;
}

0000000000000000 t f
0000000000000000 b x
```

source3.c

```
static int x;
static void f() {
    x = 3;
}

int main() { }

0000000000000000 t f
0000000000000011 T main
0000000000000000 b x
```

exe Symbols

```
0000000000400536 t f
0000000000400547 t f
0000000000400558 t f
0000000000400569 T main

0000000000601020 b x
0000000000601024 b x
0000000000601028 b x
```

- From the linker's perspective, individual .o files' symbols are either global or local
- Assembly level: default is local; must say `.globl` otherwise
- At the C level: default is global; must say `static` to make local
- Note: different use of local/global than local vs global variables. Here, "local" means local to a compilation unit, i.e., a .c file (plus headers)
- Local symbols in different compilations units are separated and do not conflict with one another or with global symbols in other units

Conflict Resolution Rules for Global Symbols

- Question: what happens if 2 or more modules define a global symbol with the same name?
- Answer: it depends on whether the symbol is considered "strong" or "weak"
  - strong + strong → conflict "multiply defined"
  - strong + weak → weak definition is ignored
  - weak + weak → one of the weak definitions is used
- These rules are a historic quirk (blame Fortran's COMMON blocks); fortunately, there is only one case in normal use that makes a symbol weak: defining an uninitialized global variable, e.g. `int x;` or `struct struct type obj;`
- This allows for the (questionable) convenience of defining the same global variable multiple times in different compilation units and have the linker turn the other way

Understanding Definitions and Declarations in C

writing	is a	that defines	and sets
Functions			
`static void f();`	declaration of f	nothing	
`static void f() { }`	definition of f	a local symbol f	
`void g();`	declaration of g	nothing	g an external ref
`extern`[1] `void g();`	declaration of g	nothing	g an external ref
`void g() { }`	definition of g	a strong global symbol g	
Variables			
`static int v;`	definition of v	a local symbol	it to 0
`static int w = 42;`	definition of w	a local symbol	it to 42
`int v;`	definition of v	a weak global symbol	it to 0
`extern int v;`	declaration of v	nothing	v an external ref
`int v = 42;`	definition of v	a strong global symbol	it to 42

---

[1]optional

Effect of Definitions and Declarations in a Header File

writing	error?	
Functions		
`static void f();`	maybe	makes sense only if defined in same header file
`static void f() { }`	no	usually ok when inlining is intended
`void g();`	no	recommended way of declaring global functions
`extern`[2] `void g();`	no	recommended way of declaring global functions
`void g() { }`	multiply-defined	strong symbol conflict rule
Variables		
`static int v;`	no	separate copies of v! Likely wrong.
`static int w = 42;`	no	separate copies of w! Likely wrong.
`int v;`	no	one copy; fragile
`extern int v;`	no	recommended way of declaring a global variable
`int v = 42;`	multiply-defined	strong symbol conflict rule

---

[2]optional

Best Practices - Variables

- Avoid global variables where possible; but if you must have them:
- Do not define global variables in a header file, regardless of static or not
  - Instead, declare them in exactly one header file (with `extern`) and choose exactly one .c file in which to define them (these files often have the same basename, as the module is said to own them)
  - Do this regardless of whether an initial value is provided for the variable
  - C++ even requires this: One Definition Rule
- Do not define non-static global variables in a .c file unless they are used in more than one .c file:
  - If multiple files define the same name, then strong definitions will conflict, weak definitions will silently refer to the same copy, as will strong/weak combinations
  - Make them static instead – maximize encapsulation
- `-Wl,--warn-common` should be quiet

VIRGINIA TECH.

Best Practices - Functions

- If not used in more than one .c file, make static and keep in .c file
- If used in more than one .c file, place prototype declaration in header file; enforce this with `-Wmissing-prototypes`
  - Do not ignore "implicit declaration" warnings
- Choose good naming scheme, such `file_` for functions in `file.c`
- Define small functions you intend for the compiler to inline in header files

Best Practices - Inline Functions

- Inlining: the compiler will insert the body of a function at the call site, avoiding procedure call overhead and enabling optimizations
- Requires that the compiler has access to the source code of the function, thus its definition in a header file; excessive use would increase compile times
- Compiler will decide whether to inline, based on chosen optimization level and on heuristics
- Which modifier should be used?
- Option 1: `static` or `static inline`. Adding `inline` is good practice, but doesn't sway or force compiler to actually inline.
- Option 2: (in C99 or later) (just) `inline` in a header file, and choose exactly one compilation unit to add an `extern inline` declaration.
- Option 2 has the advantage that it avoids multiple copies in the case where the compiler doesn't inline, but is more complicated and does not allow header-only libraries

VIRGINIA TECH.

Conclusion

- Discussed best practices for placing declarations and definitions in .c source and .h header files
- Avoid/debug linker errors and fragile practices
- Emerging alternatives: whole-program optimization techniques
  - Link-Time Optimization (LTO): compiler stores intermediate representation in .o files, optimization and code generation is done at link time on whole program
  - Concatenating the source code of multiple files (so-called "unity builds")