Linking – when you compile code and get the executable file

steps

1. The c preprocessor takes the c files (ex main.c) and translates it into main.i
2. Then the c compiler turns main.i to main.s (assembly)
3. Then the assermbler turns main.s to main.o (object file)
4. Then the linker takes all object files and converts them to a file that you can run

Why linkers?

* Modularity
  + Programs can be written as a collection of smaller source files rather than one giant code file.
  + Can build libraries of common functions ex math library, standard C library.
* Efficiency
  + Time: separate compilation
  + Space: libraries. Common functions can be aggregated into a single file, so exe files contain only code they actually use

What do linkers do?

1. Symbol resolution
   1. Programs define and reference symbols (global variables and functions)
   2. Symbol definitions are stored in object file (by assembler) in symbol table (an array of structs)
   3. During the symbol resolution step, the linker associates each symbol reference with exactly one symbol definition
2. Relocation
   1. Compilers and assemblers generate code and data sections that start at address 0
   2. Linker merges
   3. Relocates symbols
   4. Updates all references

Static linking

* Programs are translated and linked using a compiler driver

Three kinds of object files (modules)

* Relocatable object file (.o file)
  + Produced from one .c source file
* Executable object file (a.out file)
* Shared object file (.so file)
  + Called DLLs by windows

Exe and linkable format (ELF)

Linker symbols

* Global symbols: the scope is modules (any of the 3 types of object files), ex non static functions and global variables
* External symbols: defined on one module but called from other modules
* Local symbols: defined and only referenced in one module, ex anything with static attribute, ex a variable within a method inside a .c file

Local symbols

* Local non static c variables – stored on stack
* Local static c variables – stored in either .bss or .data

Program symbols are either strong or weak

* Strong – procedures and initialized globals
* Weak – uninitialized globals

STEP 1: SYMBOL RESOLUTION

Linker’s symbol rules

1. Multiple strong symbols are not allowed
   1. Each item can only be defined once
   2. Otherwise: linker error
2. Given a strong symbol and multiple weak symbols, choose the strong symbol
   1. References to the weak symbol resolve to the strong symbol
3. If there are multiple weak symbols
   1. Pick an arbitrary one
   2. Or throw an error
      1. Modern implementation of gcc throws an error, but we can overwrite this with a flag: gcc -fno-common

Linker puzzles

* Linker error because p1 (a strong symbol) is defined twice
* Idk
* Writing to x could overwrite y because x is also defined as a double, so writing to x will overwrite a double worth of memory, and since y likely uses memory after x, it will overwrite that memory

Global variables

* As you can see from the above examples in the slide, global variables are the problem
* So avoid them whenever possible
* Otherwise
  + Use static
  + Initialize them when they are defined
  + Use extern

STEP 2: RELOCATION

Relocate object files (split into text and data), then send them to executable object file

Packaging commonly used functions

* How to package functions? Math, I/O …
* Awkward given the linker framework so far:
  + Put all functions into a single source file
  + Or put each function into a separate source file
* Static libraries (.a archive files)
  + Create the library statically at linking time
* Dynamic libraries

LINKING SUMMARY

* Linking is a technique that allows programs to be constructed from multiple object files
* Linking can happen at different times in a program’s life
* Understanding linking can help avoid nasty bugs

Example program for interpositioning

* Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code
* 3 solutions: interpose on the lib malloc() and free() functions at compile time, link time, and load/run time

Compile time interpositioning

* Create a header with the same name as the standard function. (malloc.h)
* Check slide 6 for the code
* When compiling the code, we need to add an additional flag -I so that the preprocessor looks for malloc.h in the current directory

Link time interpositioning

* Use two prefixes: \_\_real, and \_\_wrap
* So we create a \_\_wrap\_malloc() function, and inside it we call \_\_real\_malloc()
* Need flags: -Wl,--wrap,malloc -Wl,–wrap,free

Load/run time interpositioning

* Use dlsym() function
* Use flags: -shared -fpic mymalloc.so