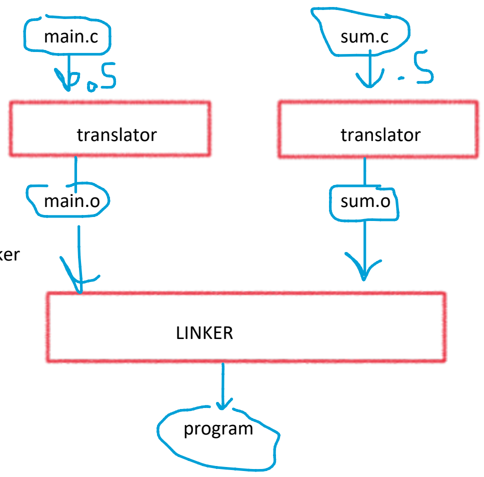
Linking

Tuesday, December 3, 2019 2:31 PM

System-level concept, not complicated from a programming perspective

there's LOTS and LOTS of code. ESPECIALLY on Operating Systems

lunch asop_arm-eng - compiles code for Android OS. Might as well go have lunch after you run this



Basic plan to make things compile faster:

Only recompile files that we modified.

- compile to an intermediate form, Object Files (*.o)
    - Assembly Snippets that can be linked together later to create a final binary
    - Each Object File represents the code for one C File
- Making a change to one C file means you only have to change one Object File
- Makefile (a type of *build system* - universal concept) handles coordination of these files

Multiple program files are smashed together into a single program, using something called a Linker
Linkers are Modular and Efficient

What do Linkers Do?

1. Symbol Resolution

programs define/reference symbols, aka names for global variables and functions
ex:

    void swap() <- defines a symbol named swap
    int *xp = &x <- defines a symbol named xp

These symbols are stored in a struct called the symbol table
during symbol resolution, the linker associates each symbol reference in the program to exactly one symbol definition
local variables are easy to resolve, because they're only referenced in the same module.

Resolving duplicate symbol definitions:

Symbols are classified as either strong or weak

- Strong: procedures, initialized globals
- Weak: uninitialized globals

Rules for resolution:

- Multiple Strong Symbols of the same name are not allowed
    - Each item can be defined only once
    - Linker Error otherwise
- Given a strong symbol and multiple weak symbols of the same name, choose the strong symbol
    - references to the weak symbol default to the strong one
- If there are multiple weak symbols, pick an arbitrary one
    - Nice
    - Can override this with "gcc -fno-common"

Because of this evil system, errors like this happen'

main.c

```
int x;
int y;
p1() {}
```

sum.c

```
double x;
p2() {}
```

Both files compile - X is an int in main.c, and x is a double in sum.c

When the linker receives this, it picks one arbitrarily to refer to the other
this means that **writes to x in sum.c might overwrite y in main.c (as well as x obviously)**

-Nightmare Scenario: two identical weak structs, compiled by different compilers with different alignment rules
-Because of this stuff, we try to avoid using global variables unless we have to

-If we have to define a global, use extern when you reference it in a different file (e.g. extern int x;)

2. Relocation

Merges separate code, data sections into single sections
relocates symbols from .o files to their final memory locations in the executable
updates references to these symbols to correspond to their new positions

Three types of object files:

- relocatable object files (.o)
- executable object files (a.out)
- shared object files (.so)

Executable and Linkable Format (ELF)

- standard binary format for object files
- handles .o, a.out, and .so files

ELF object file format:

Binary format!!!

- ELF header
  - Description of file:
  - word size, byte ordering, file type, machine type, etc.
- Segment header table
  - required for executables
  - maps sections to virtual memory addresses, so the system knows how to do this
  - Page Size, virtual address memory segments, segment sizes
- .text section
  - code (machine code, in binary)
- .rodata section
  - Read only data, printf strings, etc.
- .data section
  - initialized global/static variables
- .bss section
  - uninitialized global/static vars
  - Has section header, occupies no space (don't need values)
  - "better save space"
- .symtab section
  - Symbol table, stores defined symbols
  - procedure and static variable names. section names and locations
  - No local variables
- .rel.txt section
  - Relocation info for .txt section (the code)
  - list of locations in .txt section that have to be modified to contain the real addresses of functions/data after relocation
- .rel.data section
  - Relocation infor for .data section
  - addresses of pointer data that need to be modified, like .rel.txt
- .debug section
  - info for symbolic debugging with gcc -g
- Section Header Table
  - Offsets and sizes of each function

ELF header	
Segment header table	
.text	
.rodata	
.data	
.bss	
.symtab	
.rel .txt	
.rel .data	
.debug	
Section Header Table	

Packaging Commonly Used Functions

Commonly used functions in C such as atoi, printf, math stuff, random, etc. need to be included in compilation

Instead of including one giant file with everything (space and time inefficient), or lots or small files (annoying for us), let's compile the names of many small files into a Static Library

Static Library - the outdated solution to this

- An archive file that contains names/references to all the .o files defined by the library
- Upon compilation, the linker scans through your program and decides which files it needs to execute your stuff
- Then, the linker copies over only the .o functions from the .a file that are needed by your program
- Ordering of these libraries/.o files matters! Put all archive files last

C standard library - libc.a
C math library - libm.a

The archiver program that creates these allows incremental updates

Shared Libraries - similar to shared object files

Dynamic Linking

- The shared library is copied into physical memory, and shared among all of the running processes via virtual memory sharing
- Different processes map different virtual addresses to the same physical address
- One copy of the library at any point in time