Executables

CSE 351 Winter 2021

Instructor:

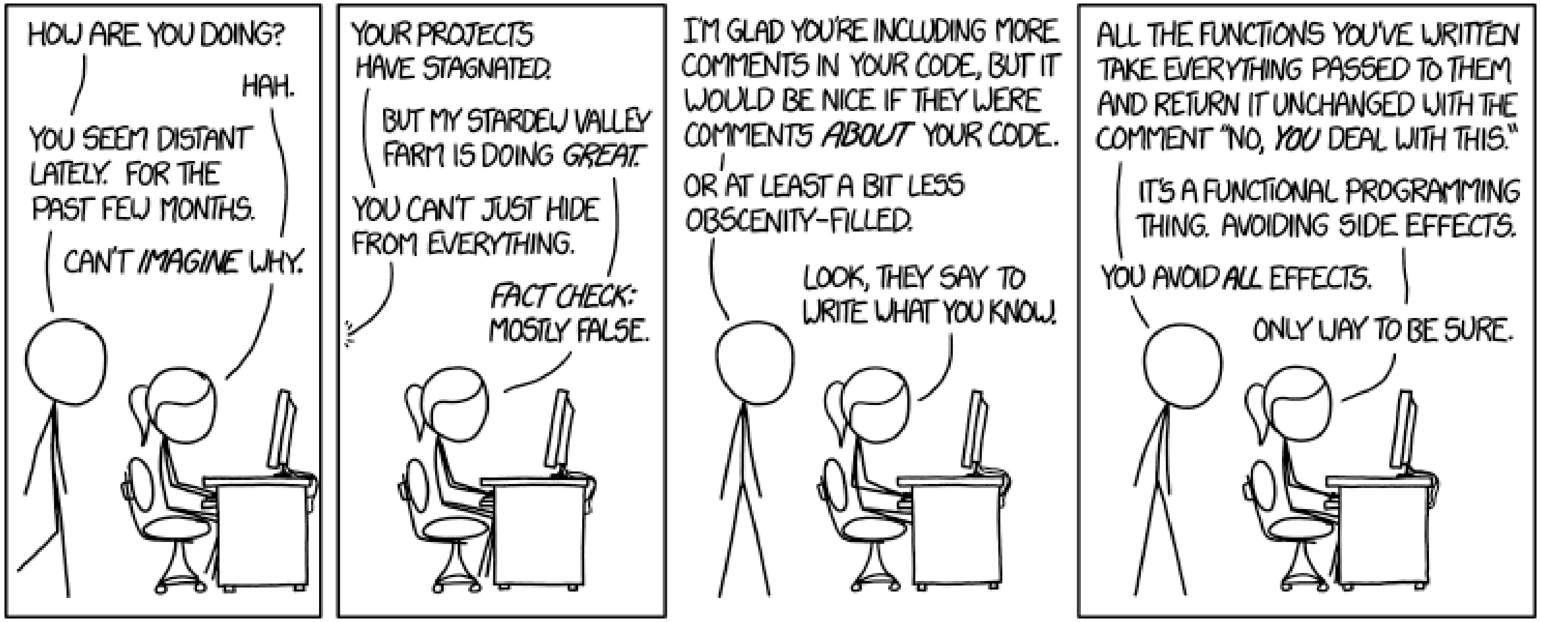
Mark Wyse

Teaching Assistants:

Kyrie Dowling Catherine Guevara Ian Hsiao

Jim Limprasert Armin Magness Allie Pfleger

Cosmo Wang Ronald Widjaja


HOW ARE YOU DOING?
HAH.
YOU SEEM DISTANT LATELY. FOR THE PAST FEW MONTHS.
CAN'T IMAGINE WHY.
YOUR PROJECTS HAVE STAGNATED.
BUT MY STARDEW VALLEY FARM IS DOING GREAT.
YOU CAN'T JUST HIDE FROM EVERYTHING.
FACT CHECK: MOSTLY FALSE.
I'M GLAD YOU'RE INCLUDING MORE COMMENTS IN YOUR CODE, BUT IT WOULD BE NICE IF THEY WERE COMMENTS ABOUT YOUR CODE.
OR AT LEAST A BIT LESS OBSCENITY-FILLED.
LOOK, THEY SAY TO WRITE WHAT YOU KNOW.
ALL THE FUNCTIONS YOU'VE WRITTEN TAKE EVERYTHING PASSED TO THEM AND RETURN IT UNCHANGED WITH THE COMMENT "NO, YOU DEAL WITH THIS."
IT'S A FUNCTIONAL PROGRAMMING THING. AVOIDING SIDE EFFECTS.
YOU AVOID ALL EFFECTS.
ONLY WAY TO BE SURE.

http://xkcd.com/1790/

Administrivia

- Lab 2 due Monday (2/8)
- hw12 due Friday
- hw13 due *next* Wednesday (2/10)
  - Based on the next two lectures, longer than normal

- Remember: HW and readings due **before** lecture, at 11am PST on due date

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

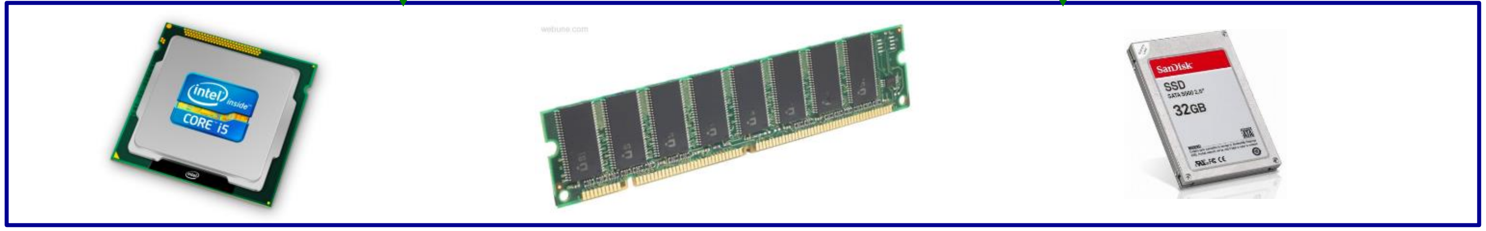
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Computer system:



- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

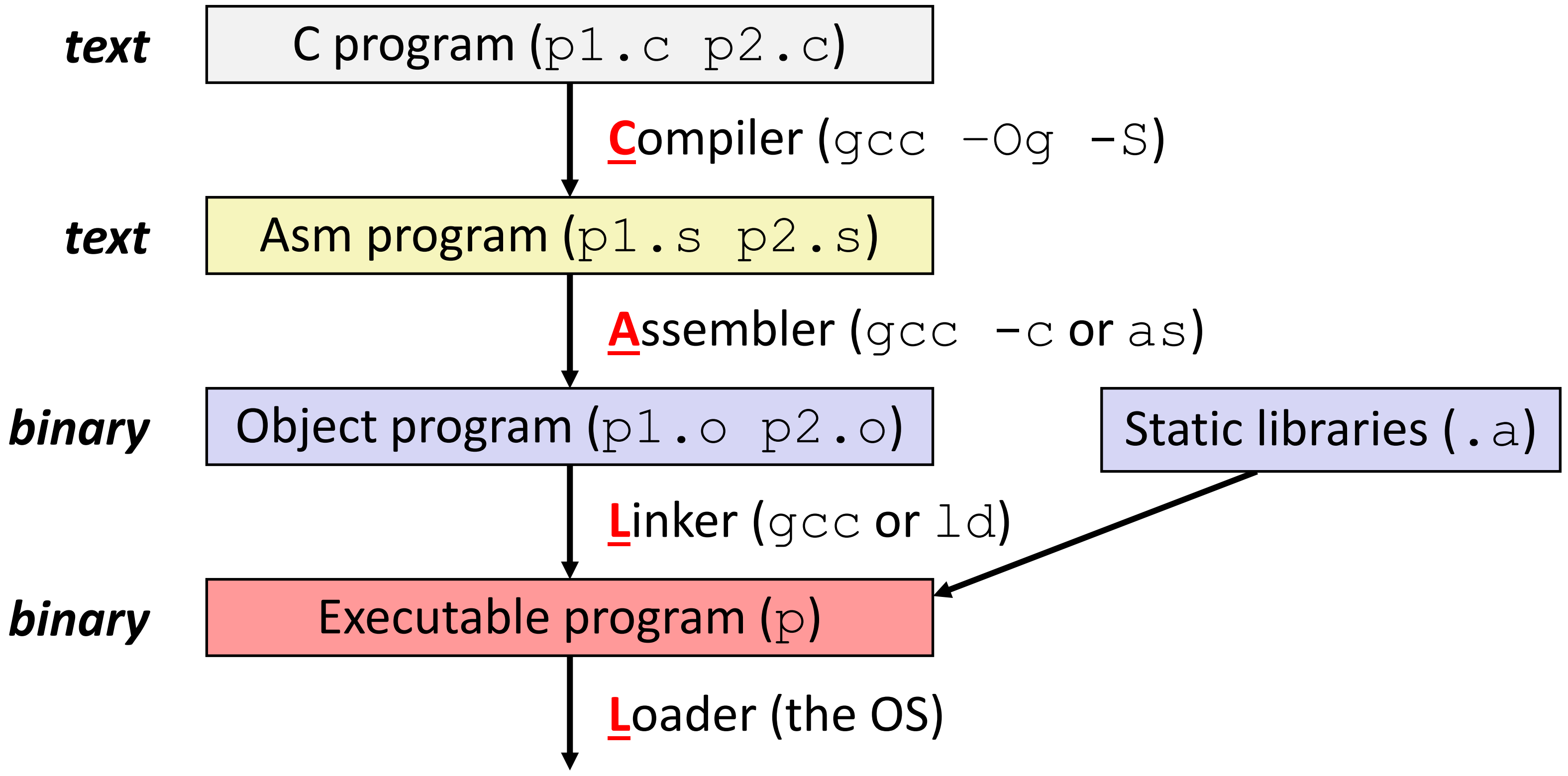
Reading Review

- Terminology:
  - CALL: compiler, assembler, linker, loader
  - Object file: symbol table, relocation table
  - Disassembly
  - Multidimensional arrays, row-major ordering
  - Multilevel arrays

- Questions from the Reading?
  - also post to Ed post!

Building an Executable from a C File

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Put resulting machine code in file `p`
- Run with command: `./p`



Compiler

- **Input:** Higher-level language code (*e.g.*, C, Java)
  - `foo.c`
- **Output:** Assembly language code (*e.g.*, x86, ARM, MIPS)
  - `foo.s`

- First there's a preprocessor step to handle #directives
  - Macro substitution, plus other specialty directives
  - If curious/interested: http://tigcc.ticalc.org/doc/cpp.html
- Super complex, whole courses devoted to these!
- Compiler optimizations
  - "Level" of optimization specified by capital '`O`' flag (*e.g.* `-Og`, `-O3`)
  - Options: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Compiling Into Assembly

- C Code (`sum.c`)

```
void sumstore(long x, long y, long *dest) {
    long t = x + y;
    *dest = t;
}
```

- x86-64 assembly (`gcc –Og –S sum.c`)

```
sumstore(long, long, long*):
  addq    %rdi, %rsi
  movq    %rsi, (%rdx)
  ret
```

Warning: You may get different results with other versions of `gcc` and different compiler settings

Assembler

- **Input:** Assembly language code (*e.g.*, x86, ARM, MIPS)
  - `foo.s`
- **Output:** Object files (*e.g.*, ELF, COFF)
  - `foo.o`
  - Contains *object code* and *information tables*

- Reads and uses *assembly directives*
  - *e.g.*, `.text`, `.data`, `.quad`
  - x86: https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html
- Produces "machine language"
  - Does its best, but object file is *not* a completed binary
- Example: `gcc -c foo.s`

Producing Machine Language

- **Simple cases:** arithmetic and logical operations, shifts, etc.
  - All necessary information is contained in the instruction itself

- What about the following?
  - Conditional jump
  - Accessing static data (*e.g.*, global variable or jump table)
  - `call`

- Addresses and labels are problematic because the final executable hasn't been constructed yet!
  - So how do we deal with these in the meantime?

Object File Information Tables

- **Symbol Table** holds list of "items" that may be used by other files
  - *Non-local labels* – function names for `call`
  - *Static Data* – variables & literals that might be accessed across files
- **Relocation Table** holds list of "items" that this file needs the address of later (currently undetermined)
  - Any *label* or piece of *static data* referenced in an instruction in this file
    - Both internal and external
- Each file has its own symbol and relocation tables

Object File Format

1) object file header: size and position of the other pieces of the object file
2) text segment: the machine code
3) data segment: data in the source file (binary)
4) relocation table: identifies lines of code that need to be "handled"
5) symbol table: list of this file's labels and data that can be referenced
6) debugging information

- More info: ELF format
  - http://www.skyfree.org/linux/references/ELF_Format.pdf

Practice Questions

- The following labels/symbols will show up in which table(s) in the object file?
  - A **(non-static) user-defined function**
  - A **local variable**
  - A **library function**

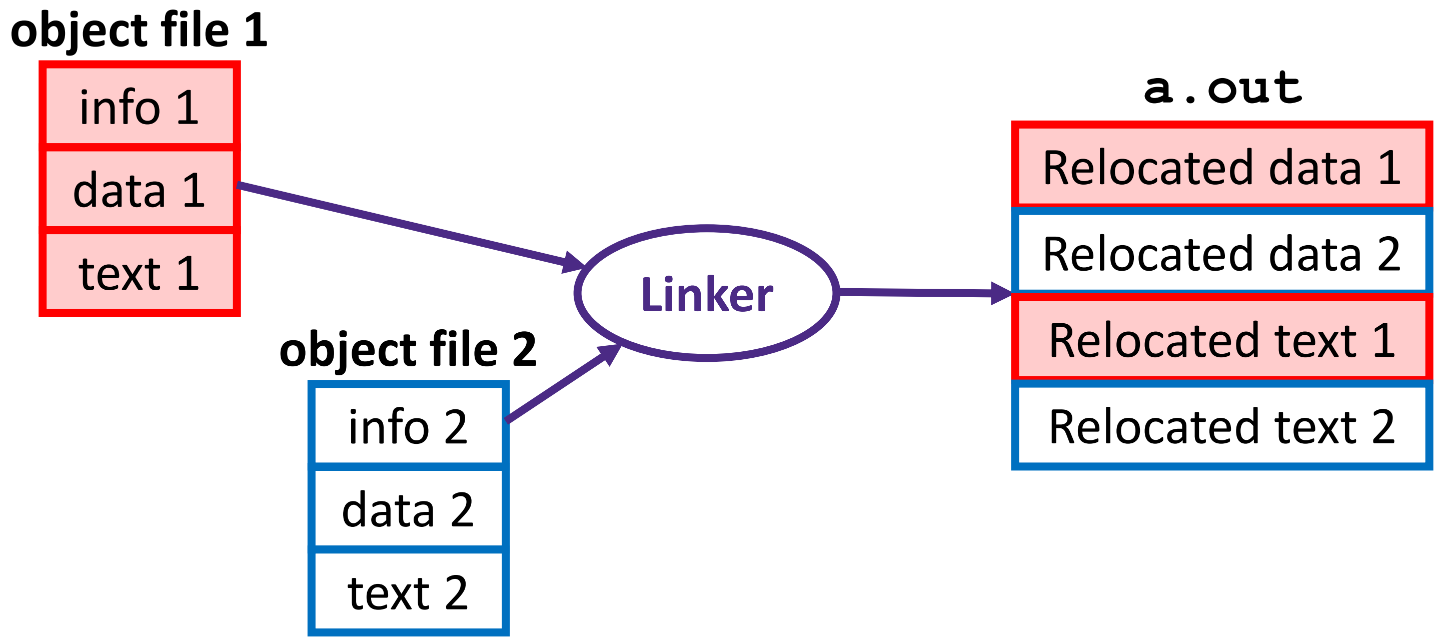
Linker

- **Input:** Object files (*e.g.*, ELF, COFF)
  - `foo.o`
- **Output:** executable binary program
  - `a.out`

- Combines several object files into a single executable (*linking*)
- Enables separate compilation/assembling of files
  - Changes to one file do not require recompiling of whole program

Linking

1) Take text segment from each `.o` file and put them together
2) Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry



Disassembling Object Code

- Disassembled:

```
0000000000400536 <sumstore>:
  400536:   48 01 fe          add    %rdi,%rsi
  400539:   48 89 32          mov    %rsi,(%rdx)
  40053c:   c3                retq
```

- **Disassembler** (`objdump -d sum`)
  - Useful tool for examining object code (`man 1 objdump`)
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can run on either `a.out` (complete executable) or `.o` file

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:    file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by Microsoft End User License Agreement

- Anything that can be interpreted as executable code
- Disassembler examines bytes and attempts to reconstruct assembly source

Loader

- **Input:** executable binary program, command-line arguments
  - `./a.out arg1 arg2`
- **Output:** <program is run>

- Loader duties primarily handled by OS/kernel
  - More about this when we learn about processes
- Memory sections (Instructions, Static Data, Stack) are set up
- Registers are initialized