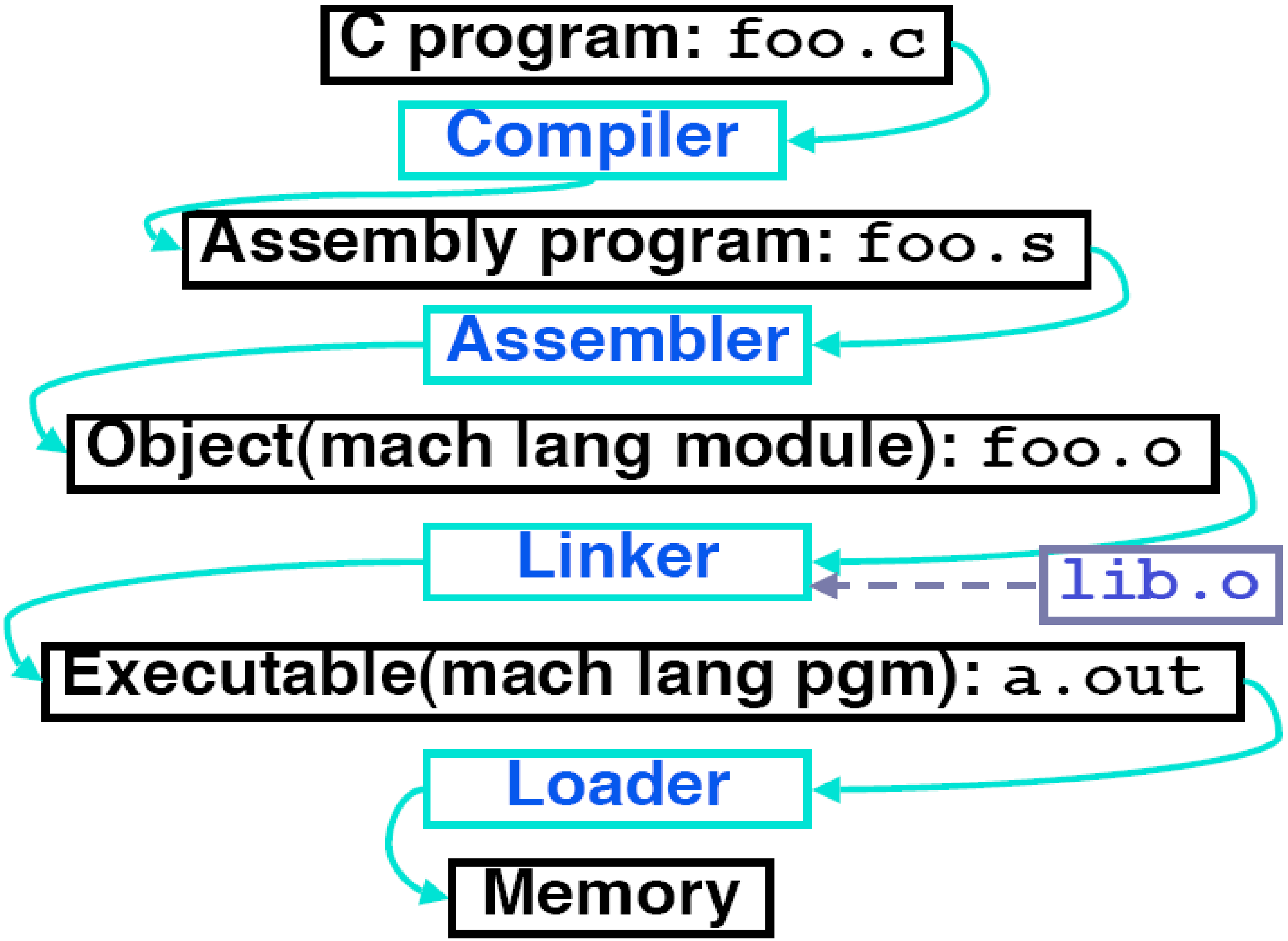
Motivation

- What happens when we type
  gcc program.c -o program?
- What work is done to turn the source code into
  something the computer can process?
- How is it possible to play N64 games on your PC?

Lowdown



C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object(mach lang module): foo.o

Linker

lib.o

Executable(mach lang pgm): a.out

Loader

Memory

Images Used with Permission
from Wikipedia

Compiler

- Translates from one programming language to another (e.g. C -> Assembly)
- Pseudo-instructions may be present in output
- Targeted optimization at this step

Assembler

- Decodes assembly language into machine language (opcodes + symbol table)
- Splits pseudo-instructions into actual ones
- Resolves symbolic names
- Processes directives
- Generates symbol and relocation tables
- Output is in an *object file*

Assembler

Regular Instructions

Pseudo-instructions

Symbolic Names

- Relocation Table
- Symbol Table

Directives

- Straight-up conversion
- Instruction: `sra $s1 $0 8`
- R Fields: 0 0 0 17 8 3
- Binary: 000000 00000 00000 10001 01000 000011
- Hex: 0x00008A03
- What about `la $a0, str`?

Assembler

Regular Instructions

Pseudo-instructions

Symbolic Names

- Relocation Table
- Symbol Table

Directives

- `la $a0, str`
- Pseudo, so broken down into:

  -
    ```
    lui $at,left_16(str)
    ori $a0,$at,right_16(str)
    ```
- Note the usage of `$at`

Assembler

Regular Instructions

Pseudo-instructions

Symbolic Names

Relocation Table

Symbol Table

Directives

-
  ```
  L1: slt  $t0, $0,  $a1
      beq  $t0, $0,  L2
      addi $a1, $a1, -1
      j L1
  L2: add  $t1, $a0, $a1
  ```
- We can resolve branches right now!

-
  ```
  L1: slt  $t0, $0,  $a1
      beq  $t0, $0,  2
      addi $a1, $a1, -1
      j L1
  L2: add  $t1, $a0, $a1
  ```

Assembler

Regular Instructions

Pseudo-instructions

Symbolic Names

- Relocation Table
- Symbol Table

Directives

- Jumps are more troublesome
  - Require the absolute address
  - Don't know how objects will arrange
  - Forward addressing
- Thus, maintain two tables
  - Symbol Table
  - Relocation Table

Assembler

Regular Instructions

Pseudo-instructions

Symbolic Names

Relocation Table

Symbol Table

Directives

- Relocation table contains a list of things we need to fix in the file
  - Labels referenced in jump instructions
  - Global labels targeted by la
  - External labels not in the current file
- We fix these later on when the information becomes available

Assembler

Regular Instructions

Pseudo-instructions

Symbolic Names

Relocation Table

Symbol Table

Directives

- Symbol table associates identifiers with where it is declared
  - Relative address of labels to the start of the text segment
  - Ordering of variables in the .data segment
- This is for later reference by the program or by another object file

Assembler

Regular Instructions

Pseudo-instructions

Symbolic Names

Relocation Table

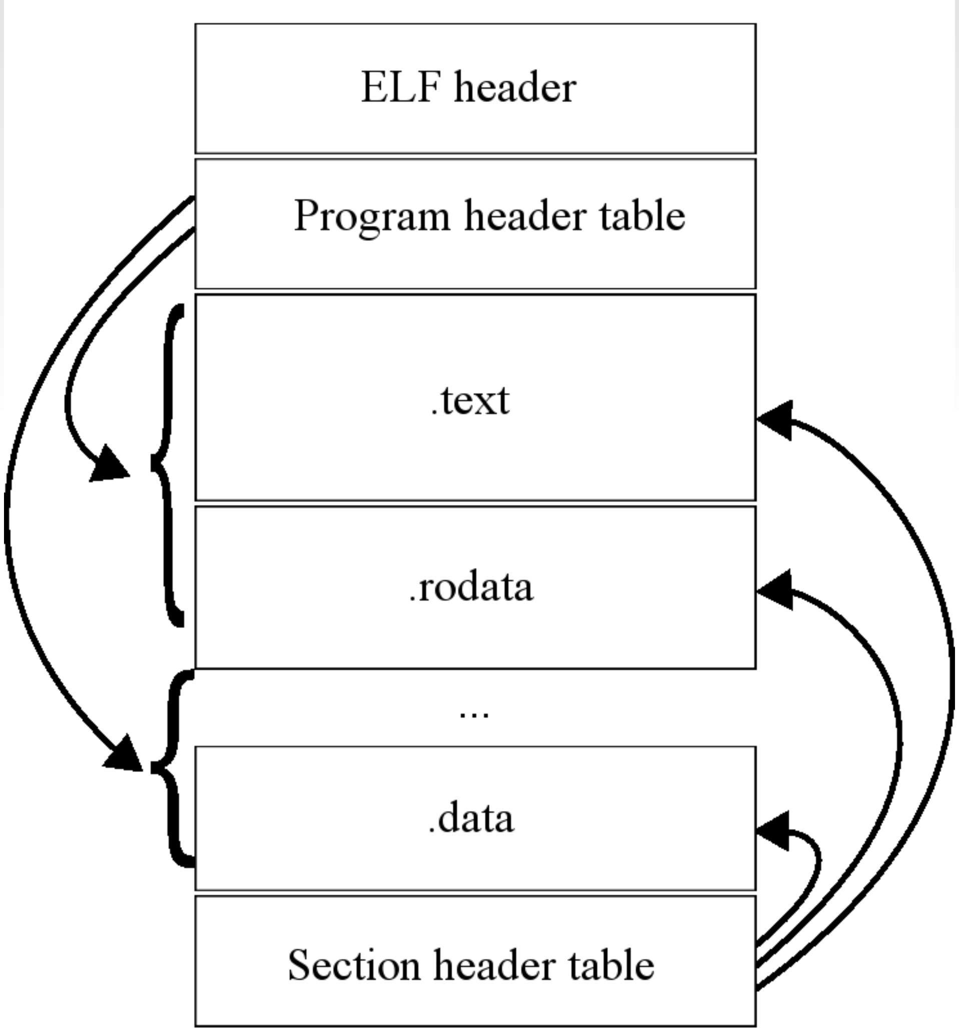
Symbol Table

Directives

- Directives give guidelines to assembler
- `.globl` – Declares a global variable
- `.text` – Marks beginning of the code segment
- `.data` – Start of variable declaration for storage in memory
- `.asciiz` – Declares a \0 terminated string
- `.word` – 32 bit integer
- And many more...

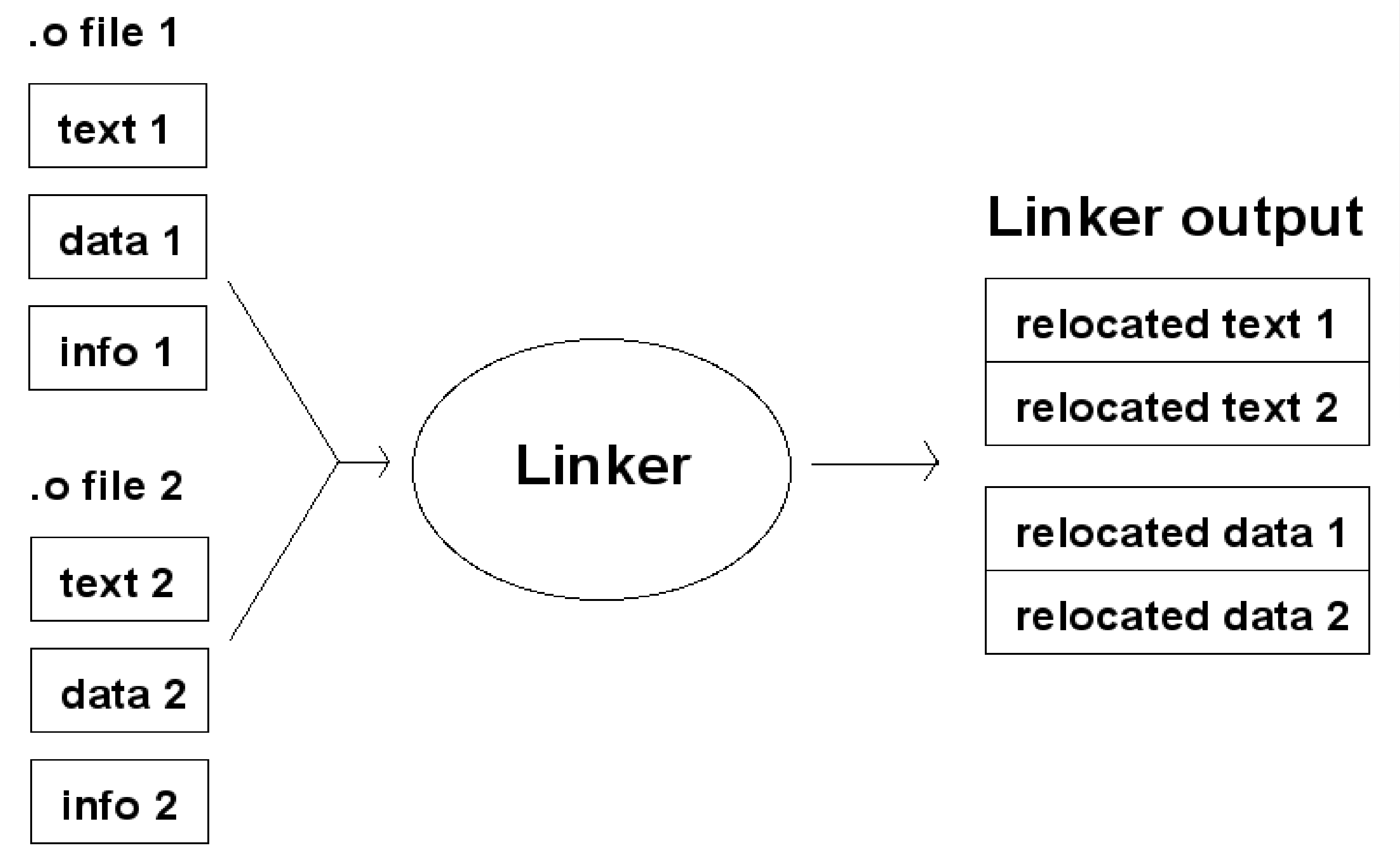
ELF Layout

- The Executable and Linking Format
  - Object Header
  - Section Headers
  - .text segment
  - .data segment
  - Relocation Table
  - Symbol Table
  - Misc. + Debug Info



Images Used with Permission from Wikipedia

Linker



Images Used with Permission from Wikipedia

Linker

- Input has all the necessary files
- Start by concatenating data and text segments
- Then fix up things in all the relocation tables by consulting the symbol table of each file
- Assume 0 is the start address of text segment
- Separate object files allow
  - Distribution of obfuscated object+header files
  - Quick recompilation of just the modified source files

Loader

- Loads text and data of program into memory
- Initialize registers ($sp, $gp, $fp, etc.)
- Moves command line input parameters to registers
- Executes and increments program counter

Summary

- Compiler turns source code into assembly code (.c -> .s)
- Assembler turns assembly code into machine code (.s -> .o)
- Linker combines many object files and libraries into an executable (.o + .o -> a.out)
- Loader loads the program into memory and runs (./a.out)

Dynamic vs Static Linking

- So far, we've done static linking
  - Embeds libraries and objects in a single file
- Dynamic linking loads required objects at runtime
  - (+) Reduced file size and memory usage at runtime
  - (+) Library updates are propagated automatically
  - (-) Linking process takes time