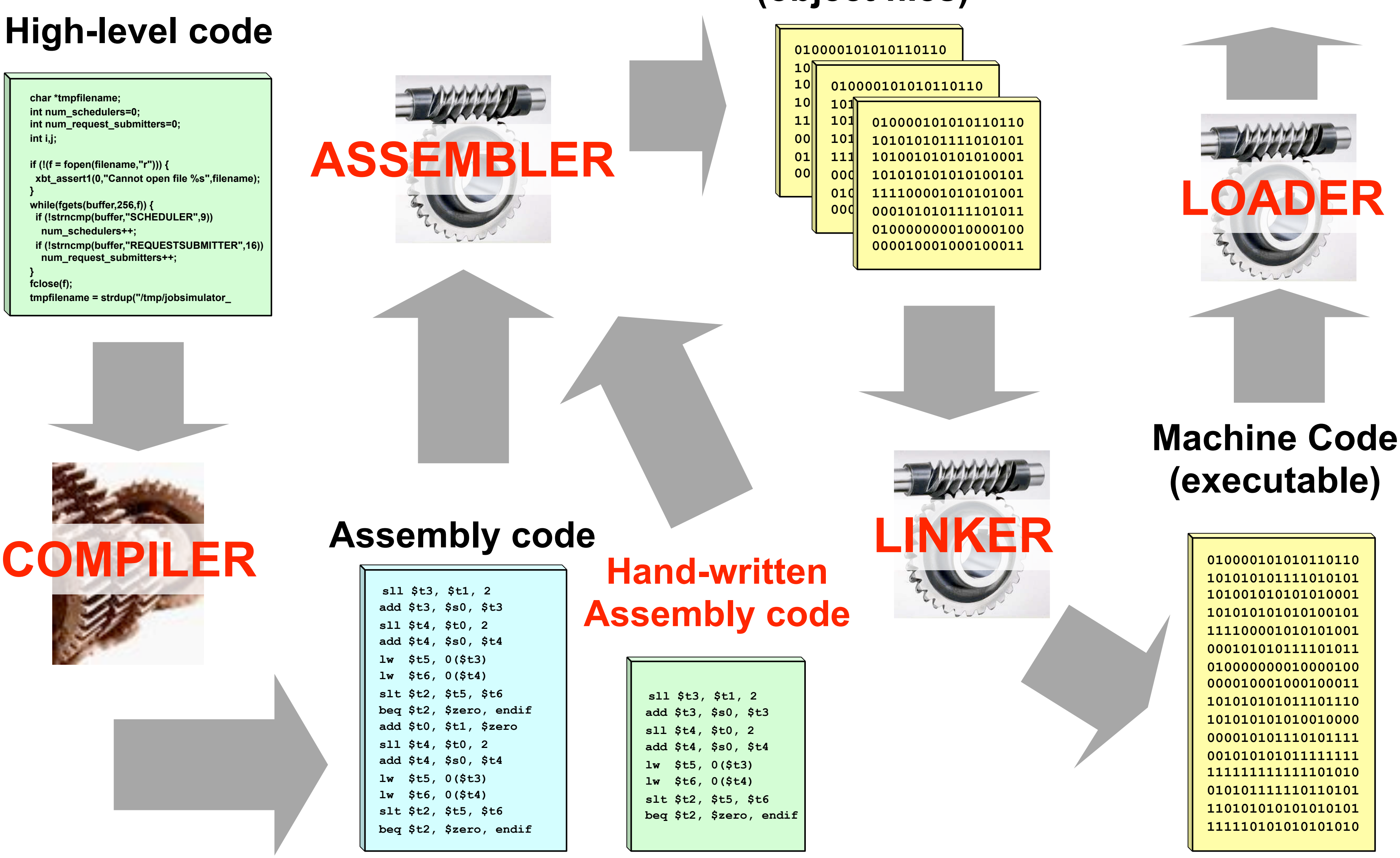
Linking and Loading

ICS312

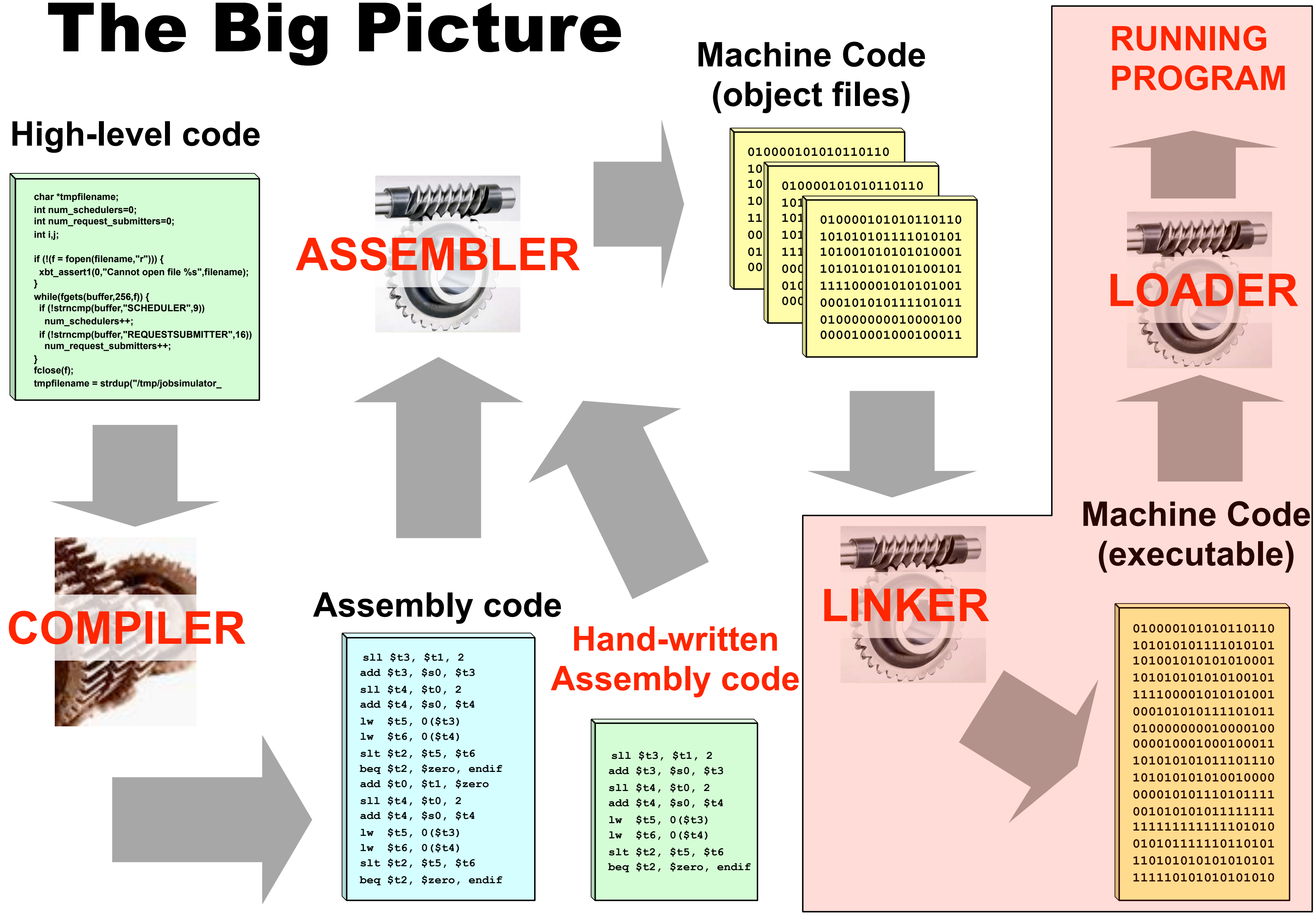
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

The Big Picture



The Big Picture



The Linker and the Loader

- You've used these two programs without really knowing it
  - We link using the "gcc" command, which calls the linker for us
    - "gcc" also calls the compiler
  - We run a program by just typing the executable name in a Shell, the Shell calls the loader for us
- In these slides we look at what these two programs do
- But first let's understand a little bit more about the structure of an object file

Object Files

- The Assembler (e.g., NASM) produces a binary object file for each .asm file
- Most assembly instructions are easily translated into machine code using a one-to-one correspondence
- But in our program we declared labels for addresses
  - Addresses in the .bss and the .data segments
  - Addresses in the .text segments (for jumps)
- Question: How should the assembler translate instructions that use these labels into machine code?
  - E.g., add   [L], ax
  - E.g., call   my_function
- Answer: it cannot do the full job without knowing the “whole” program so as to determine addresses
- Instead it just creates two tables to keep track of these names that will need to be replaced by addresses at some point

Symbol Table

- The Symbol table records the list of “items” in the file that can be used by the code in this file and in other files
  - E.g., subprograms
  - E.g., “global” variables in the data segment
- Each entry in the table contains the name of the label and its offset within this object file
- In NASM, these symbols must be declared using the global keyword
  - e.g., global   asm_main

Relocation Table

- The Relocation table records the list of "items" that this file needs (from other object files or libraries)
  - E.g., functions not defined in this file's text segment
  - E.g., "global" variables not defined in this file data segment

Object File Format

- An object file contains the following information:
  - A header that says where in the files the sections below are located
  - A (concatenated) text segment, which contains all the source code (with some missing addresses)
  - A (concatenated) data segment (which combines all data and the bss segments)
  - Relocation Table: identifies lines of code that need to be "fixed"
  - Symbol Table: list of this file's referencable" labels
  - Perhaps debugging information (is compiled with -g from a high-level programming language)
    - Source code line numbers, etc.
- There are many different specific formats, and all specifications are available on-line

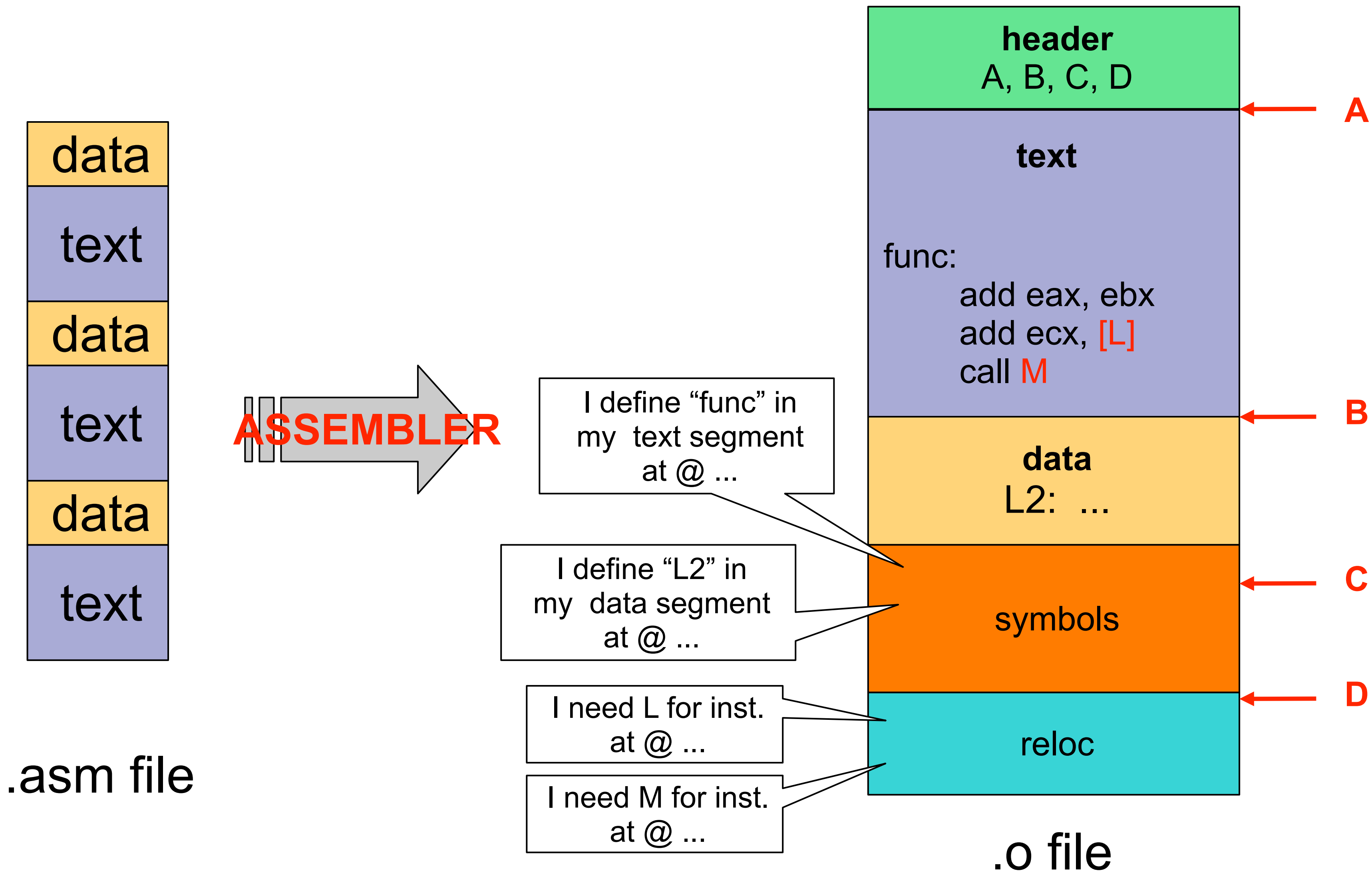
Objdump

- On Linux, the objdump command makes it possible to examine the content of an object file
- Let's try objdump on a simple C code
  - **`gcc -m32 -c objdump_demo.c -o objdump_demo.o`**
- Finding out information about different sections
  - **`objdump -h objdump_demo.o`**
    - .data, .bss, .text
    - .comment: created by gcc with version string
      - **`objdump -s --section .comment objdump_demo.o`**
    - .note.GNU-stack: empty section created by gcc to indicate that the stack doesn't need to be executable (Great to prevent buffer overflow exploit)
    - .eh_frame: used for exceptions (C++)

Objdump

- Disassembling:
  - Going from binary to assembly
  - **`objdump -d objdump_demo.o`**
  - If you know assembly, then you can try to reverse engineer code for which you only have the executable...
- Looking at the symbol table:
  - **`objdump  -t objdump_demo.o`**
- Looking at the rellocation table:
  - **`objdump  -r objdump_demo.o`**
- The “nm” program gives you table informations
  - **`nm objdump_demo.o`**

Assembling/Linking Process



The Linker

- What the linker does: combined several object files into a single executable
- This is really useful to enable separate compilation
  - You can recompile only one of your 100 .asm files, and call the linker, without recompiling all your code
    - A Makefile will use this capability
- Let us look at a simplified view of what the linker does

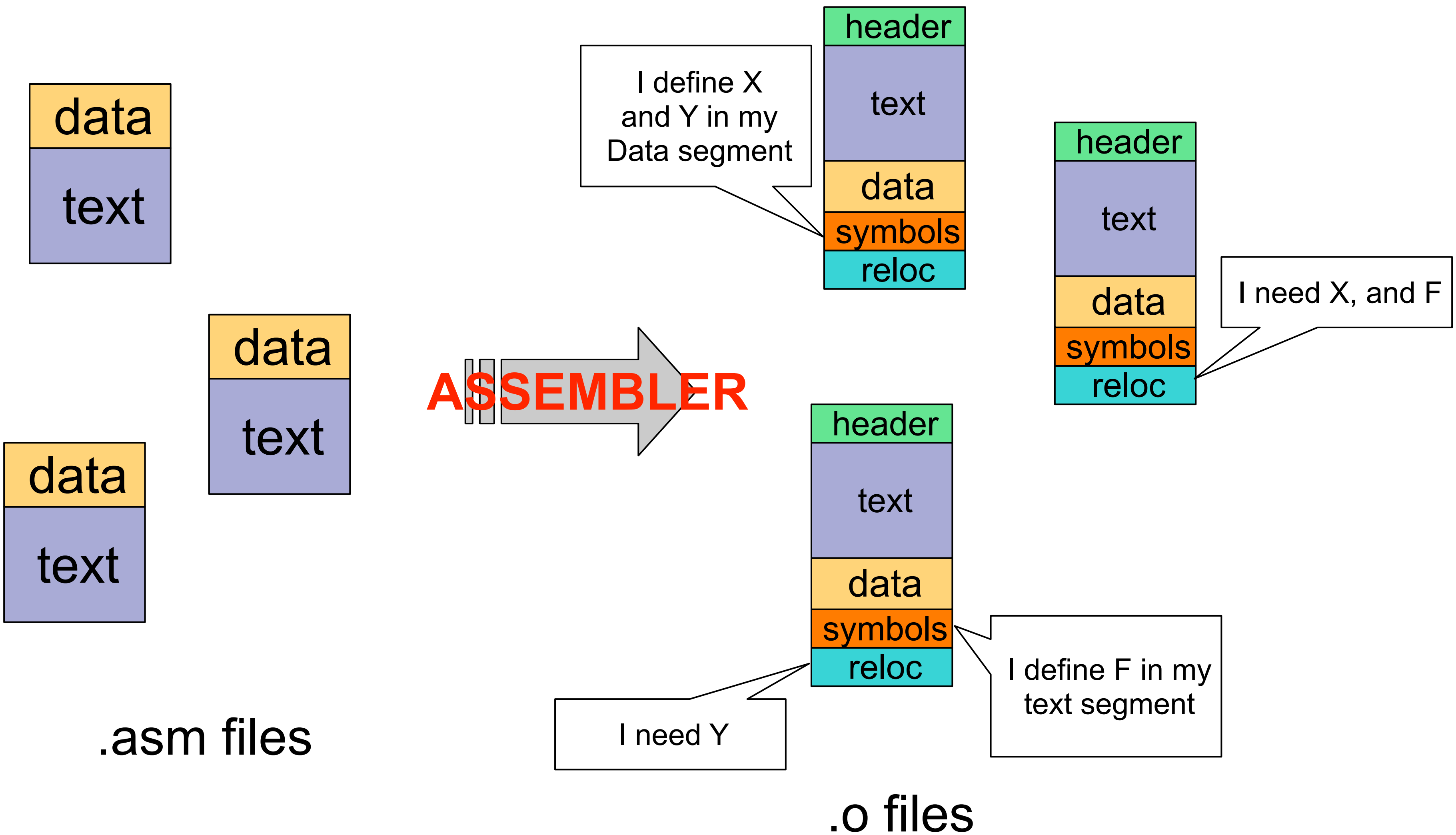
The Linker

- The linker proceeds in 3 steps
  - Step 1: concatenate all the text segments from all the .o files
  - Step 2: concatenate all the data/bss segments from all the .o files
  - Step 3: Resolve references
    - Use the relocation tables and the symbol tables to compute all absolute addresses

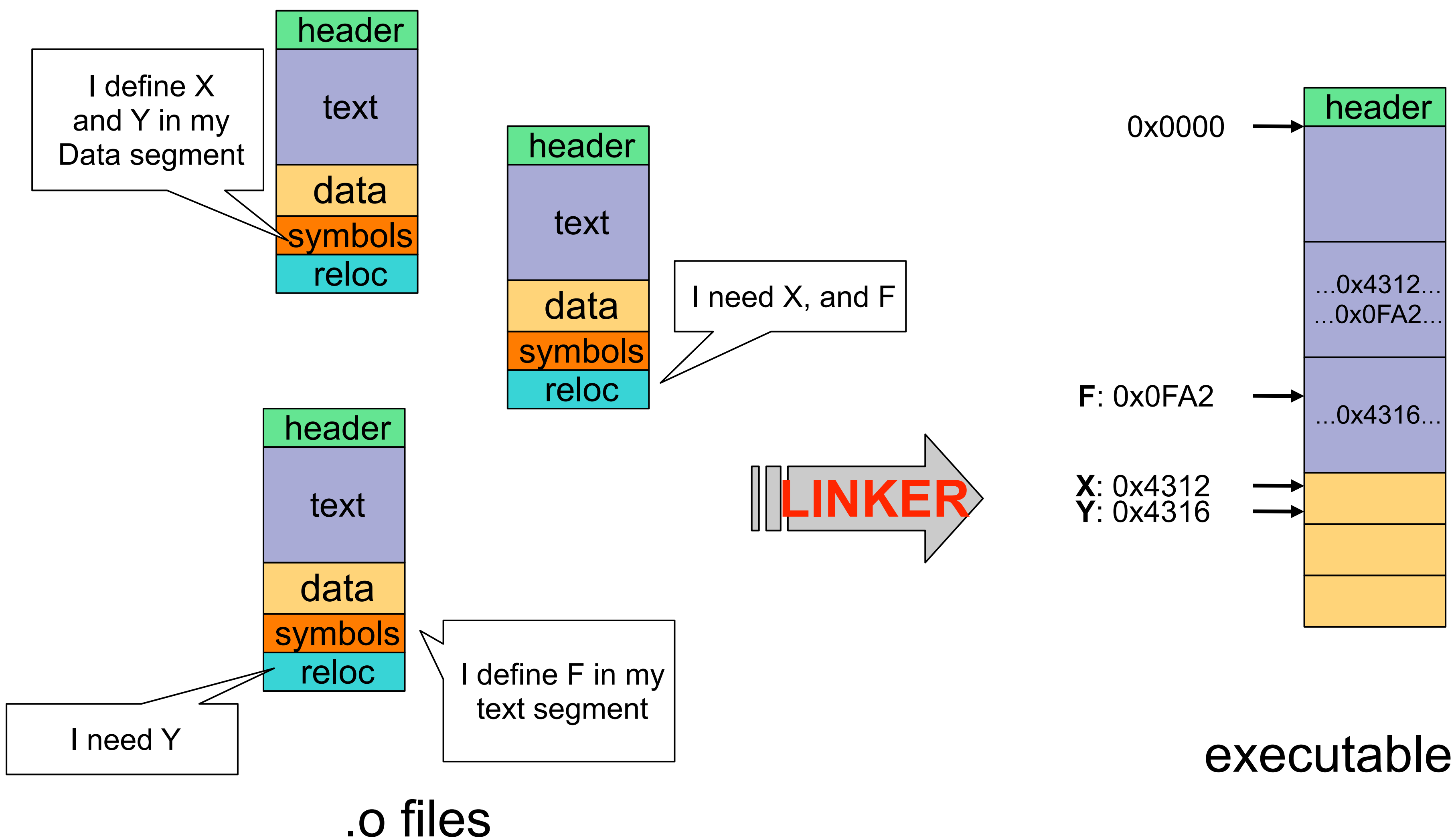
Resolving References

- The linker knows
  - The length of each text and data segment
  - The order in which they are
- Therefore the linker can compute an absolute address for each label
  - assuming the beginning of the executable file is at address 0
- For each label being referenced (that is for each line of code that's pointed to by the relocation table), find where it is defined
  - In the symbol table of a .o file
  - In some specified or standard library file (e.g., fprintf)
- If not found, print a "symbol not found" error message and abort
- If found in multiple tables, print a "multiply defined" error message and abort
- If found in exactly one table, replace the label by an absolute address
- Done when the executable file contains only absolute addresses

Assembling/Linking Process



Assembling/Linking Process



Gcc does a lot of work

- When you call gcc to compile/link your code on a Linux system, it calls many other programs
- Two well-known examples are:
  - The C Preprocessor: cpp
  - The Linux linker: ld
- The Preprocessor handles all the macros:
  - #define
  - #include
  - #if
  - . . .
- It's easy to call it by hand and see what the code really looks like before it is passed to the compiler
  - Let's try it

Gcc calls the linker

- Calling the linker by hand proves difficult because we have to give it all the object files that contain symbols that are used in the program
  - This includes all sorts of libraries that we never see when just using gcc
- Let’s try to compile a small program running “gcc -v”
  - Which shows how gcc calls ld
  - And we’ll see that in fact it calls another program called collect2

The Loader

- Now we have a linked executable, with all addresses known so that the program can run
- To actually run the program we need to use a loader, which is part of the O/S
- The loader does the following:
  - Read the executable file's header to find out the size of the text and data segments
  - Creates a new address space for the program that is large enough to hold the text and data segments, and to hold the stack (within some bounds)
  - Copies the text and data segments into the address space
  - Copies arguments passed to the program on the stack
  - Initializes the registers
    - Clear most of them, set ESP to the top of the stack
  - Jump to a standard "start up routine", which sets the PC and calls the exit() system call when the program terminates

Conclusion

- A lot of things happen under the cover when you do: gcc main.c -o main; ./main
  - Call the preprocessor
  - Call the compiler
  - Call the assembler
  - Call the linker
  - Call the loader
- You'll find out more about the sort of things the loader does in an Operating Systems class (ICS 332)