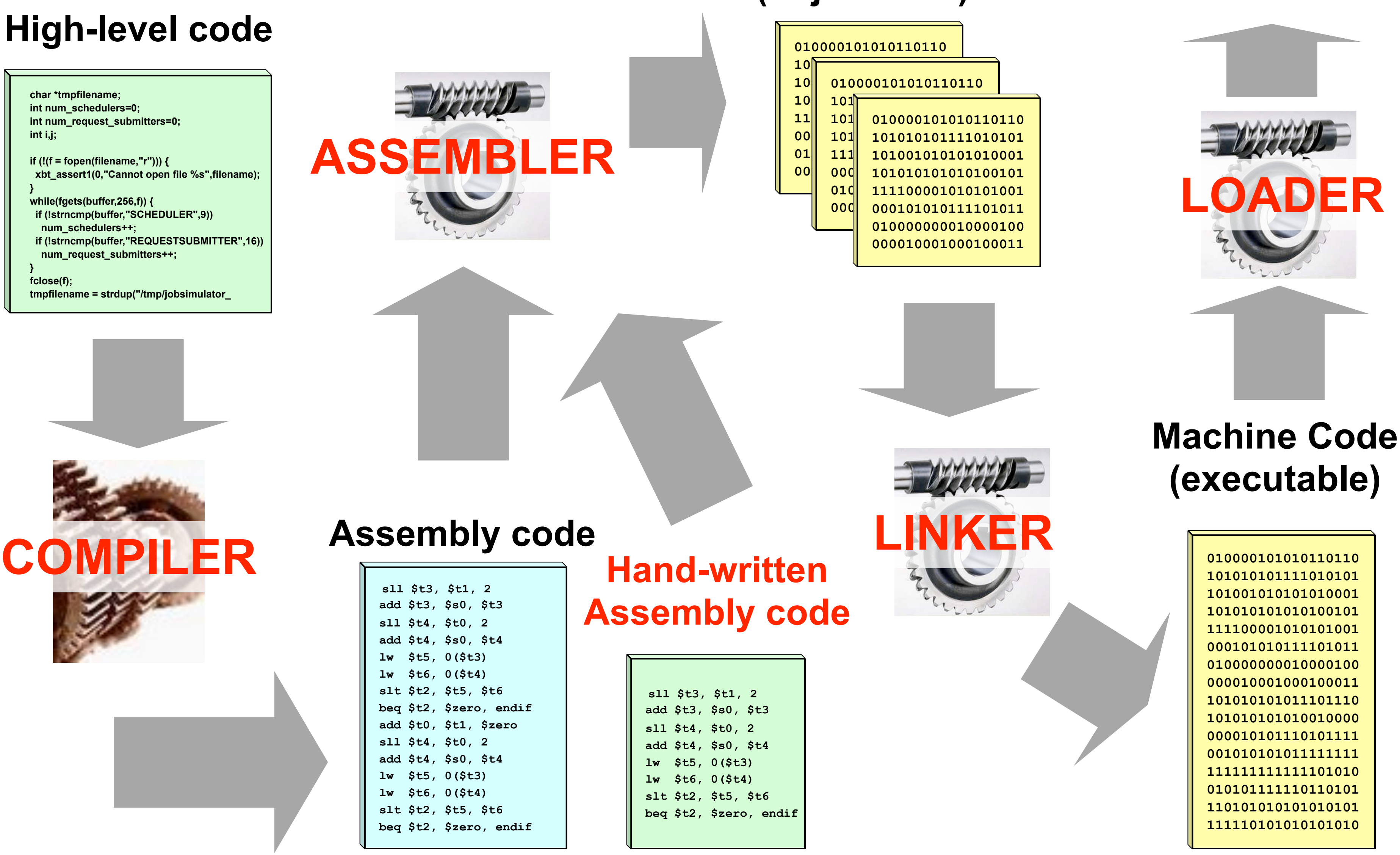
Linking

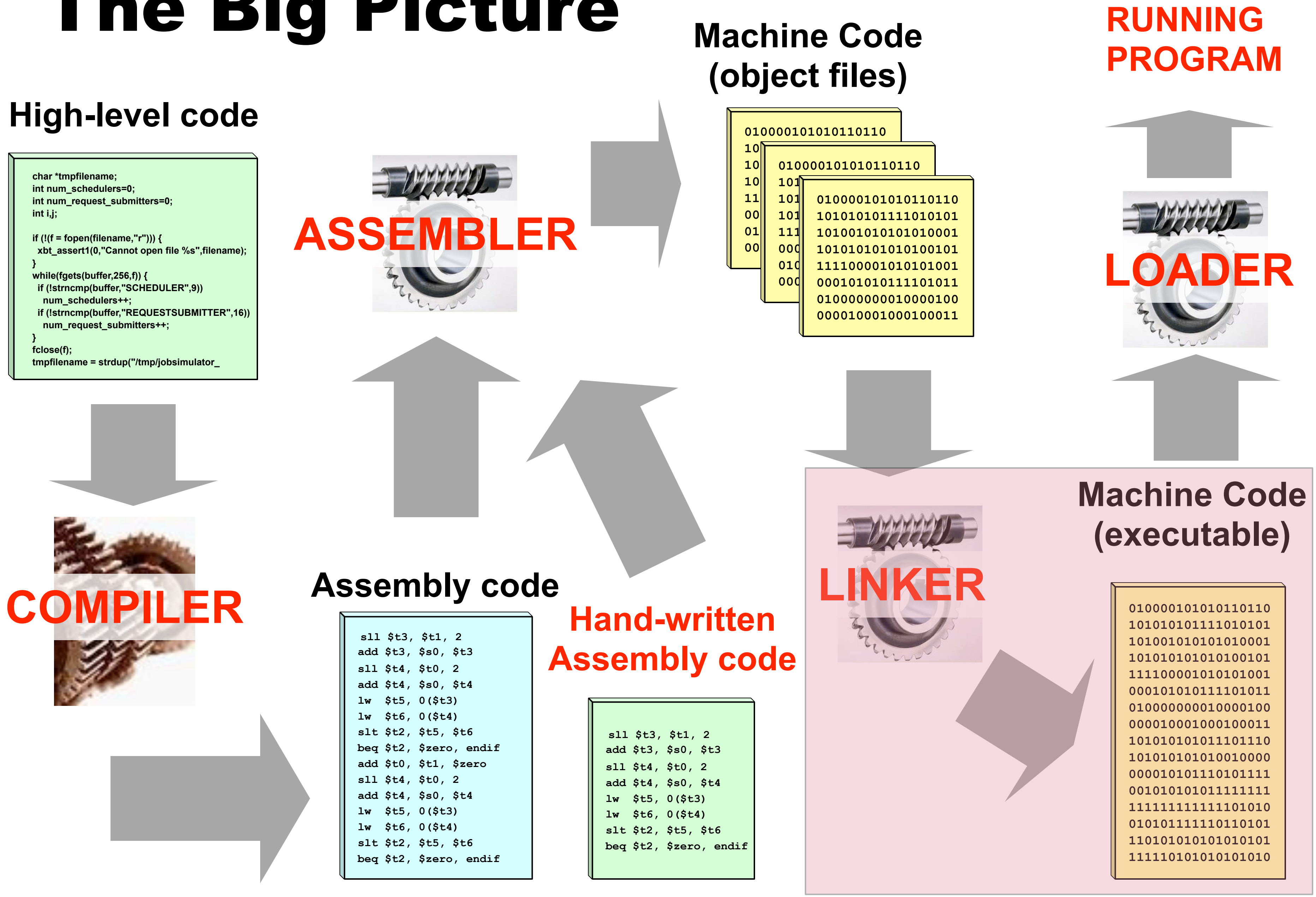
ICS312
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

The Big Picture



The Big Picture



The Linker

- You've used this program before perhaps without knowing it
  - The compiler and linker commands often look the same for convenience
    - e.g., the "gcc" command can compile and link
  - Your IDE calls the compiler/linker for you
- The principles behind linking are not complicated but first we need to understand a little bit more about the structure of an object file
  - We will not look at details of a particular system as there are a lot of them

Object Files

- The Assembler produces binary object files
- 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, .data, and .text segments
- Question: How should the assembler translate instructions that use these labels into machine code?
  - E.g., add [L], ax 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

Symbol Table

- The Symbol table records the list of “items” that the file provides and can be used by code 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
- There is one entry per places in the code where a missing reference needs to be fixed
- e.g., if a file doesn’t define function f() and contains 10 calls to f(), then it’s relocation table has 10 entered

Object File Format

- An object file contains the following information:
  - A header: says where in the file the sections below are located
  - A (concatenated) text segment: contains all the source code (with some missing addresses)
  - A (concatenated) data segment: contains all data and bss segments
  - Relocation Table: lists places in the code that need to be "fixed" because of missing addresses
  - Symbol Table: list of this file's "referenceable by others" addresses
  - Perhaps debugging information (if compiled with -g from a high-level programming language)
- There are many different specific formats, and all specifications are available on-line

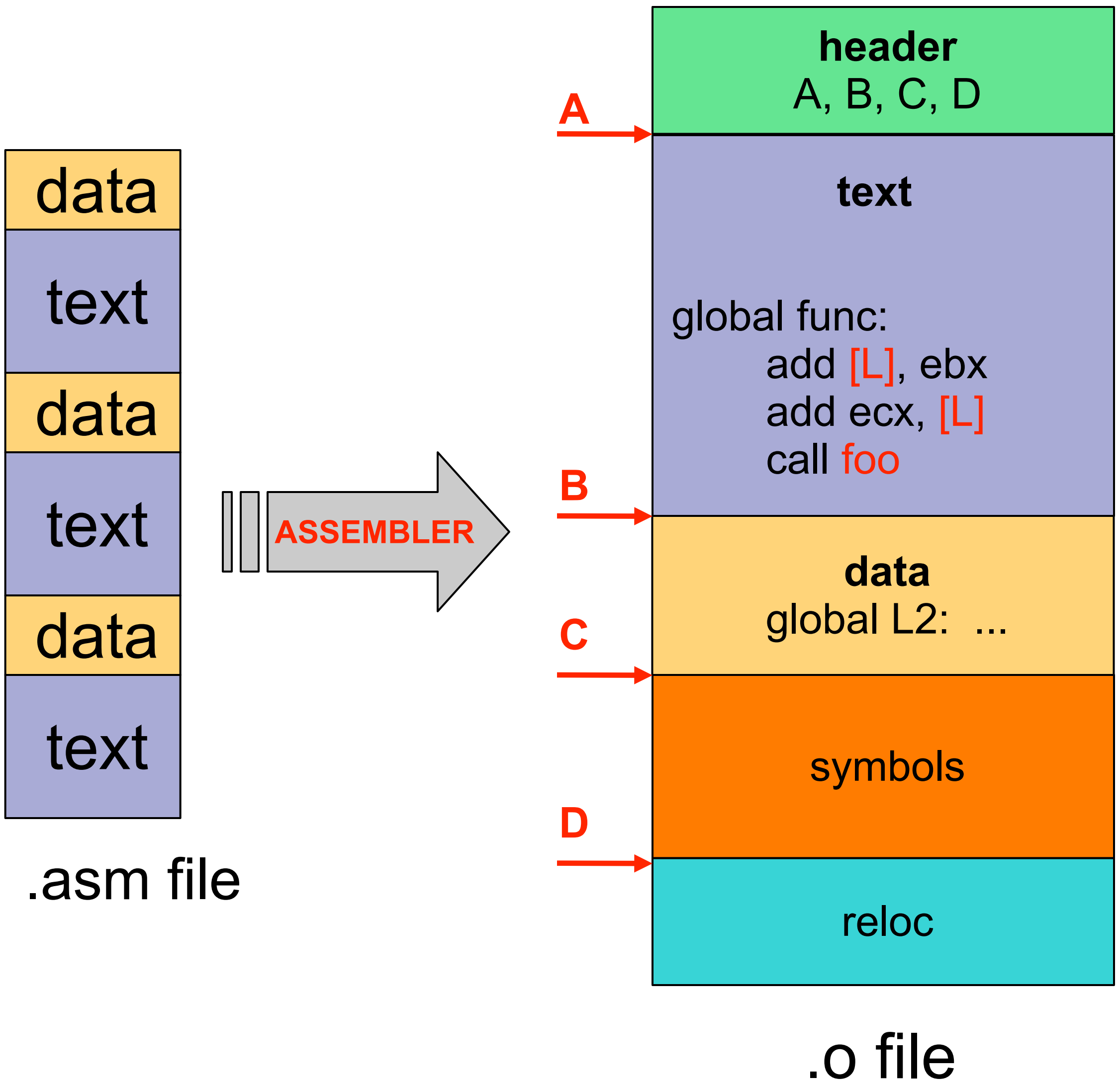
Objdump

- 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 on Linux
  - **`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++)

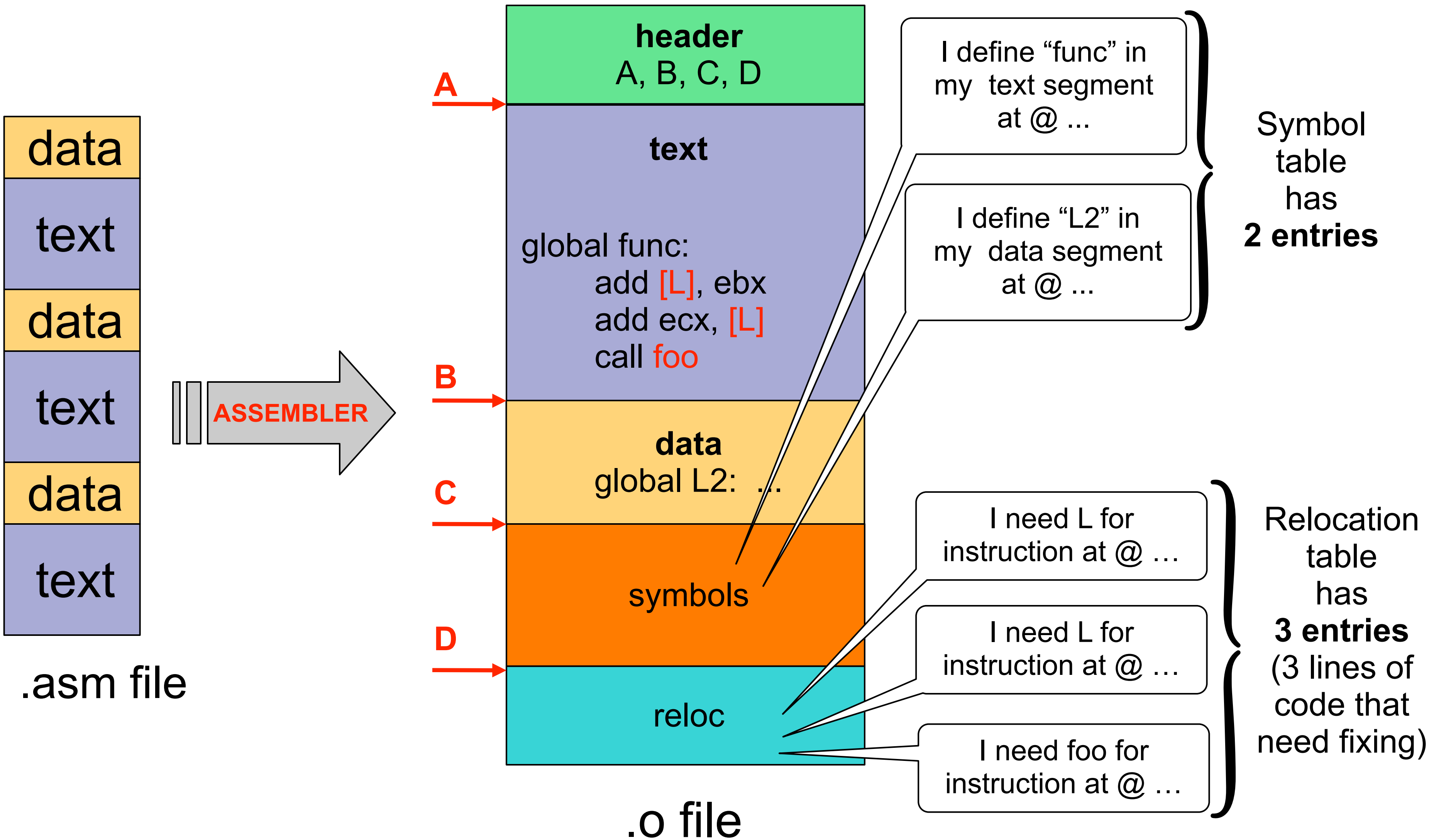
Disassembling with objdump

- Disassembling:
  - Going from binary to assembly
  - **`objdump -d objdump_demo.o`**
  - Shows ATT syntax
  - To see Nasm syntax: **`ndisasm objdump_demo.o`**
- 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



Assembling/Linking Process



Assembling/Linking Process

- What the linker does: combines several object files into a single executable
- This is really useful to enable separate compilation
  - You can recompile only one of your 100 source files, and call the linker, without recompiling all your code
  - Any self-respecting build framework will do this
- Let us look at a simplified view of what the linker does

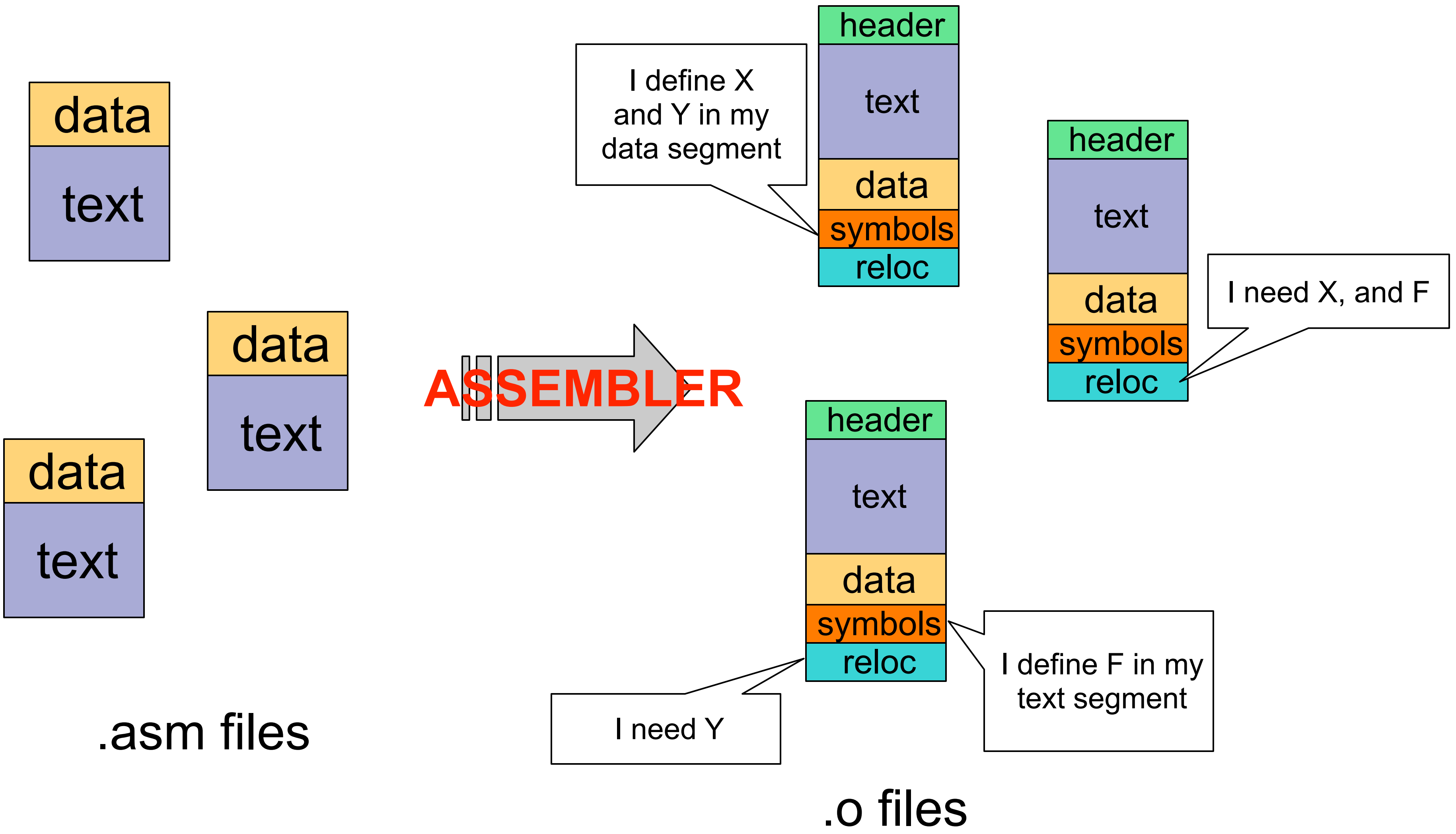
The Linker's Three Steps

- 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

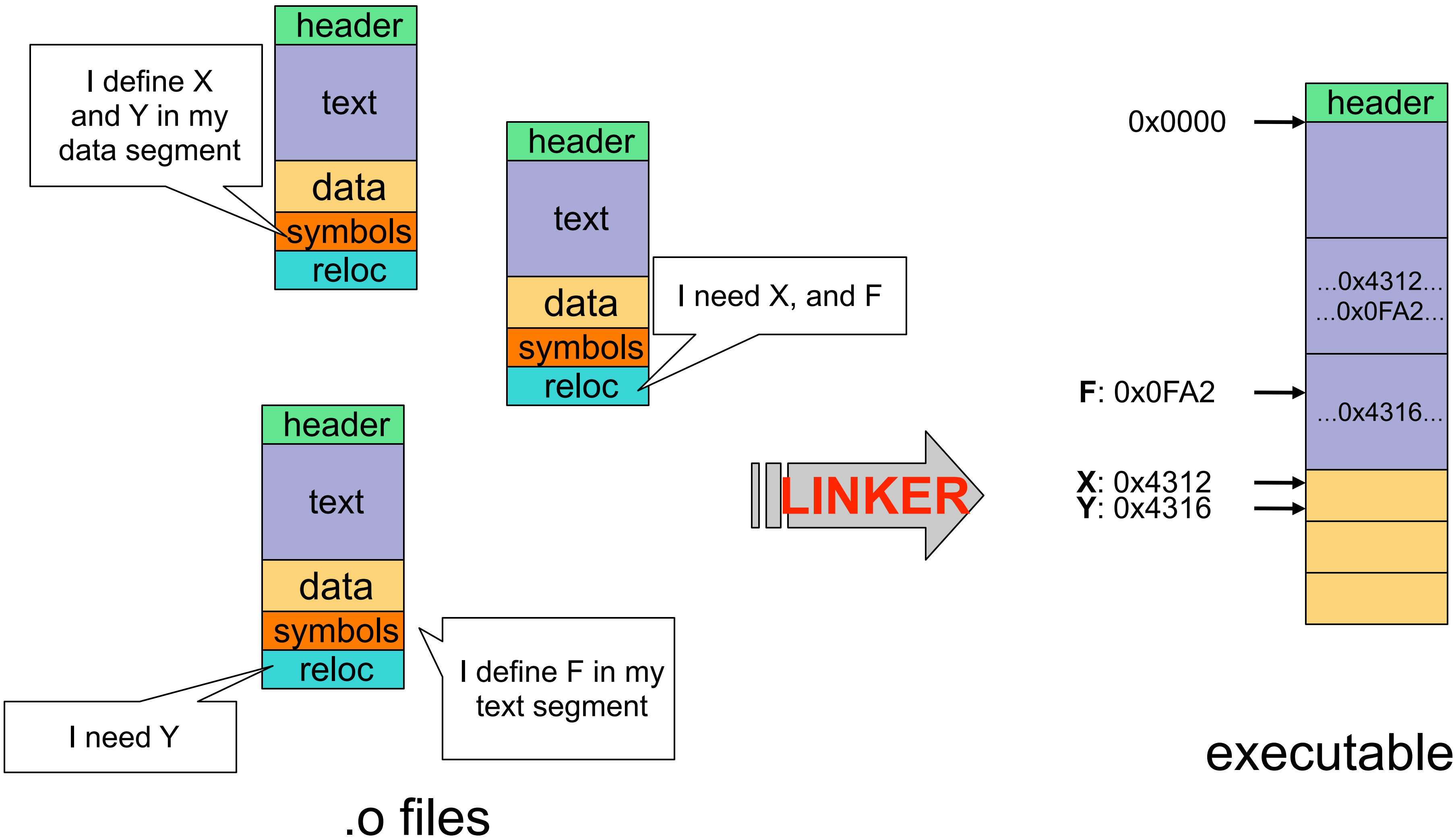
Resolving References

- The linker knows
  - The length of each text and data segment
  - The order in which they are
- Therefore the linker computes 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



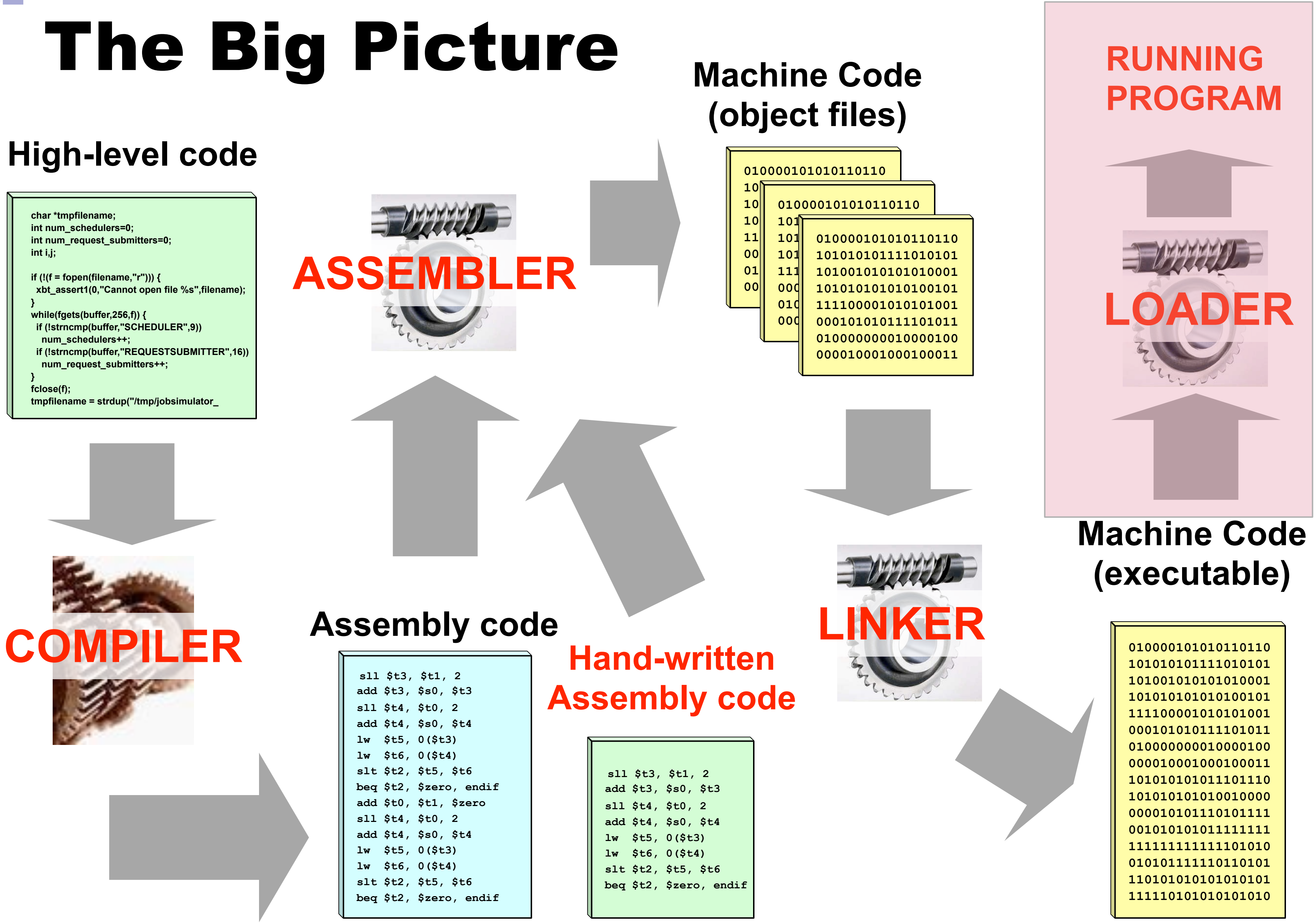
Gcc does a lot of work

- 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?
- Preprocessing is useful in many contexts, and there are generic pre-processors
  - gpp, m4, …

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

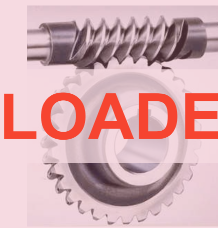
The Big Picture



The Big Picture

- The Loader is really part of the OS code
  - “in the Kernel”
- You have seen / will see this in ICS 332

RUNNING PROGRAM



High-level code

```
char *tmpfilename;
int num_schedulers=0;
int num_request_submitters=0;
int i,j;

if (!(f = fopen(filename,"r"))) {
  xbt_assert1(0,"Cannot open file %s",filename);
}
while(fgets(buffer,256,f)) {
  if (!strncmp(buffer,"SCHEDULER",9))
    num_schedulers++;
  if (!strncmp(buffer,"REQUESTSUBMITTER",16))
    num_request_submitters++;
}
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_
```

Assembly code

```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
add $t0, $t1, $zero
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```

Hand-written Assembly code

```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```

Machine Code (executable)

```
01000010101010110110
10101010101111010101
10100101010101010001
10101010101010100101
11110000101010101001
00010101010111101011
01000000000010000100
00001000100010001100
10101010101011101110
10101010101010010000
00001010101110101111
00101010101011111111
11111111111111101010
01010111111110110101
11010101010101010101
11111010101010101010
```

Conclusion

- A lot of things happen under the cover when you do: gcc main.c -o main
  - Call the preprocessor
  - Call the compiler
  - Call the assembler
  - Call the linker
- Take ICS332 to understand what happens after, i.e., how programs run
- If you take ICS312 and ICS332, then you should be able to tell a very long story if somebody asks: I have a text file that contains the string “print 12”, what are the steps so that 12 ends up printed?
  - This could literally take 30 minutes of explanations