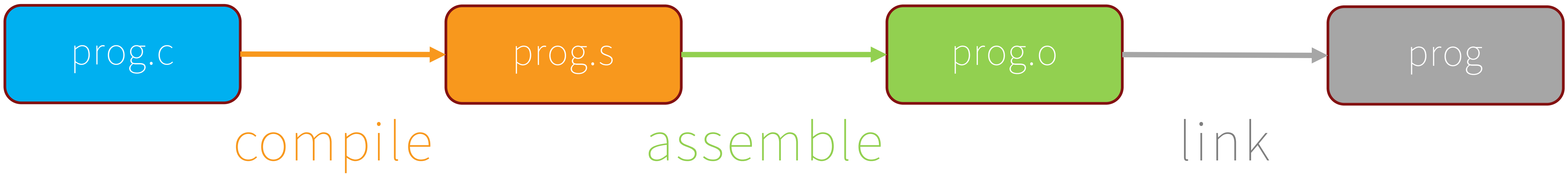
Compilation – Assemblers, Linkers, & Loaders

CS 3410: Computer System Organization and Programming

Cornell Bowers CIS
Computer Science

[K. Bala, A. Bracy, E. Sirer, and H. Weatherspoon]

Compiling – From **C** to an **Executable**

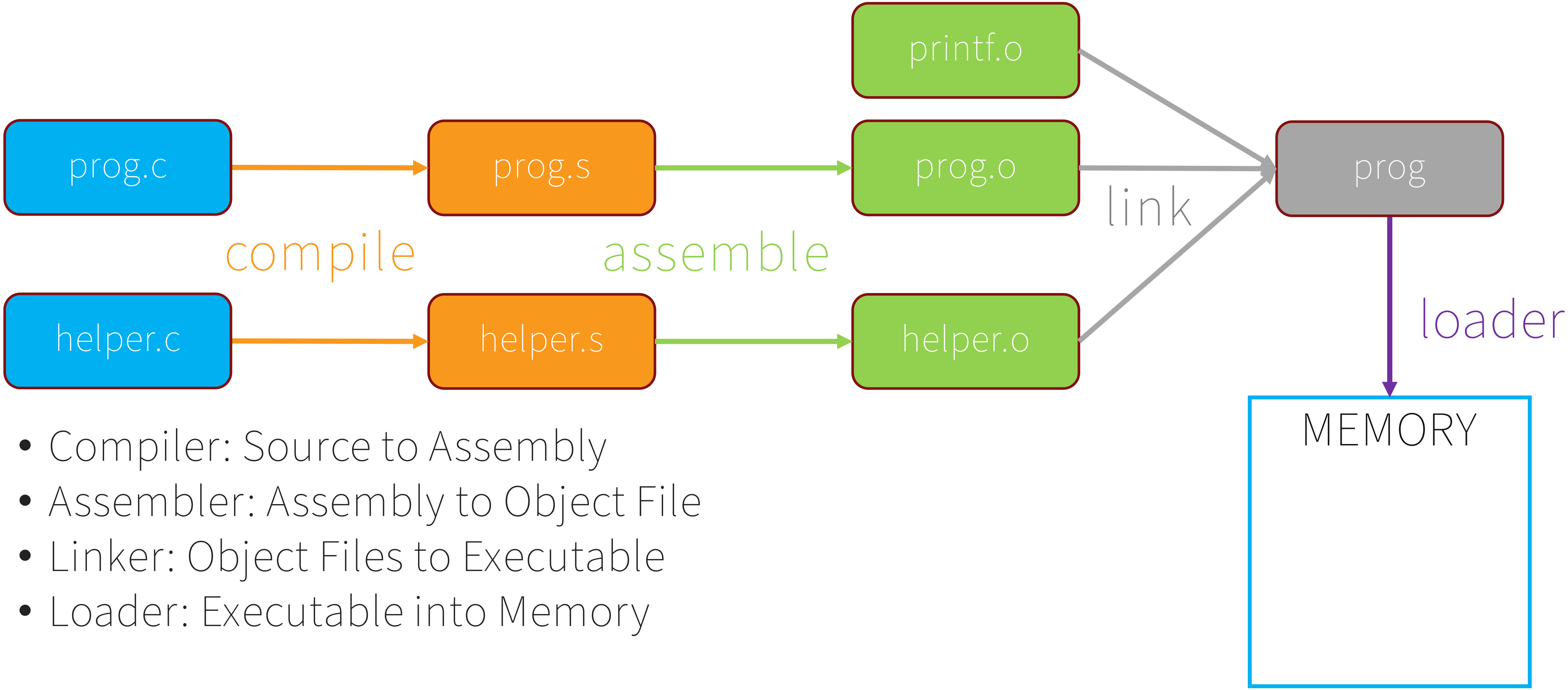


- People saying "compile" usually mean: compile + assemble + link
- It's what happens when you run:

```
gcc -o prog prog.c helper.c
```

- Last Step = "Load" program into memory (i.e., running it)

Compiling – From **C** to an **Executable**



- Compiler: Source to Assembly
- Assembler: Assembly to Object File
- Linker: Object Files to Executable
- Loader: Executable into Memory

Why The Gory Detail?

- Goal for the Course
  - Understand, from “top to bottom” what happens when your program runs on a computer
- Debug errors you *will see* as a programmer
  - Building low level code
  - Making builds ”portable”
- Efficiency of Builds
  - What tradeoffs you can make while compiling/linking to save on:
    - Space, compilation time, program efficiency

Working Example – prog.c

prog.c

```
#include <stdio.h>
#include “helper.h”

int n = 5;

int main() {
  int i = sum(n, a);
  int j = inc(i);
  printf("%d+1 = %d\n",i,j);
}
```

helper.h

```
extern int a;
int inc(int n);
int sum(int i, int j);
```

helper.c

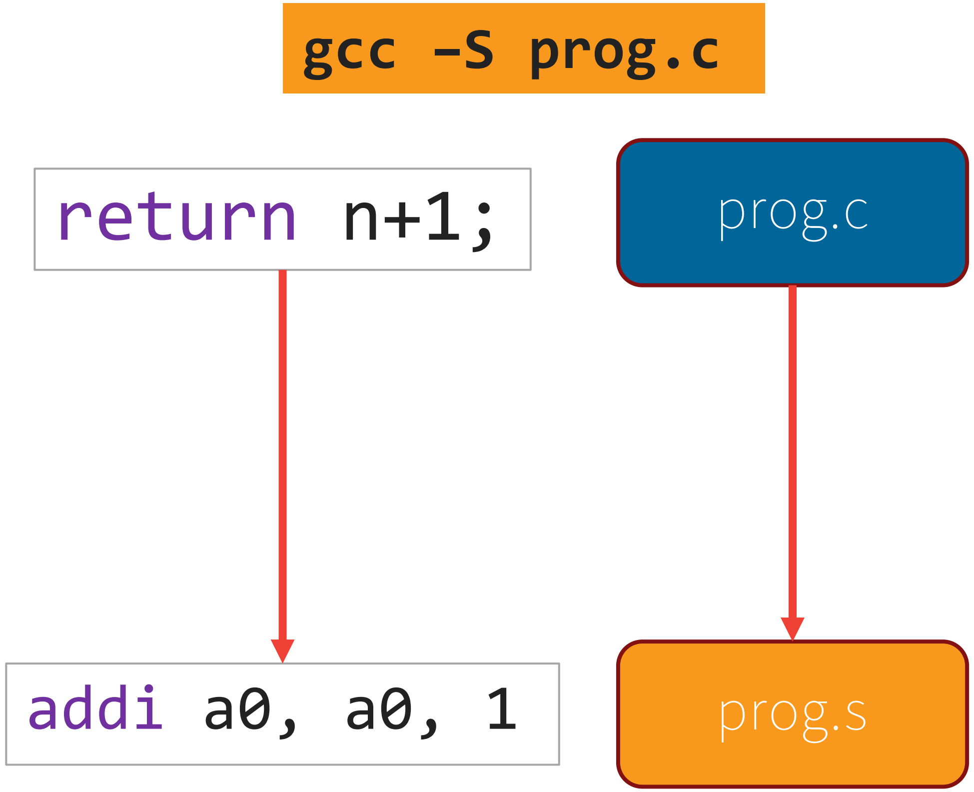
```
int a = 3;
int inc(int n) { return n+1; }
int sum(int i, int j) { return i + j ; }
```

Compiler

- Input: *.c
  - Source Code
  - Headers (function & global variable definitions)

- Output: *.s
  - Target Architecture (e.g., RISC-V, x86_64)
  - Assembly Instructions (*not yet machine code*)

(subject of the calling conventions lectures)



Compiling – prog.s contents

- Metadata
  - filename, debug symbols
- Memory layout
  - Section, alignment
- External References
  - .comm a, 4, 4 (common symbol a)
- Constants & Function Bodies
  - Still references *global and external variable names*

(you don't need to memorize these –just get a feel for what's in the assembly file)

```
.file "prog.c"
.align 2
.globl main
.type main, @function

main:
        addi sp, sp, -32
        lui a5, %hi(n)
        …
        .globl n
        …
n:
        .word   5
        .section        .rodata
        .align  3
.LC0:
        .string "%d+1 = %d\n"
        .text
        .align 2
…
```

Compiling – Procedure

- Each file compiled separately
  - (so we would also produce “helper.s” in our previous example)
- Optimizations
  - Flags: -O0, -O1, -O2, -O3 (none to all)
  - Dead code elimination, constant folding, loop unrolling, etc.
  - Some rarely (if ever) applied without programmer hints (function inlining & loop unrolling)
  - List of gcc’s optimiztions - can also flag individual opts
  - Most are *local optimizations* (i.e., functions are optimized individually)
- Take 4120 if you want to *really really* know how compilers work

```
gcc -O2 -S prog.c
```

Optimization Tradeoffs

prog.c

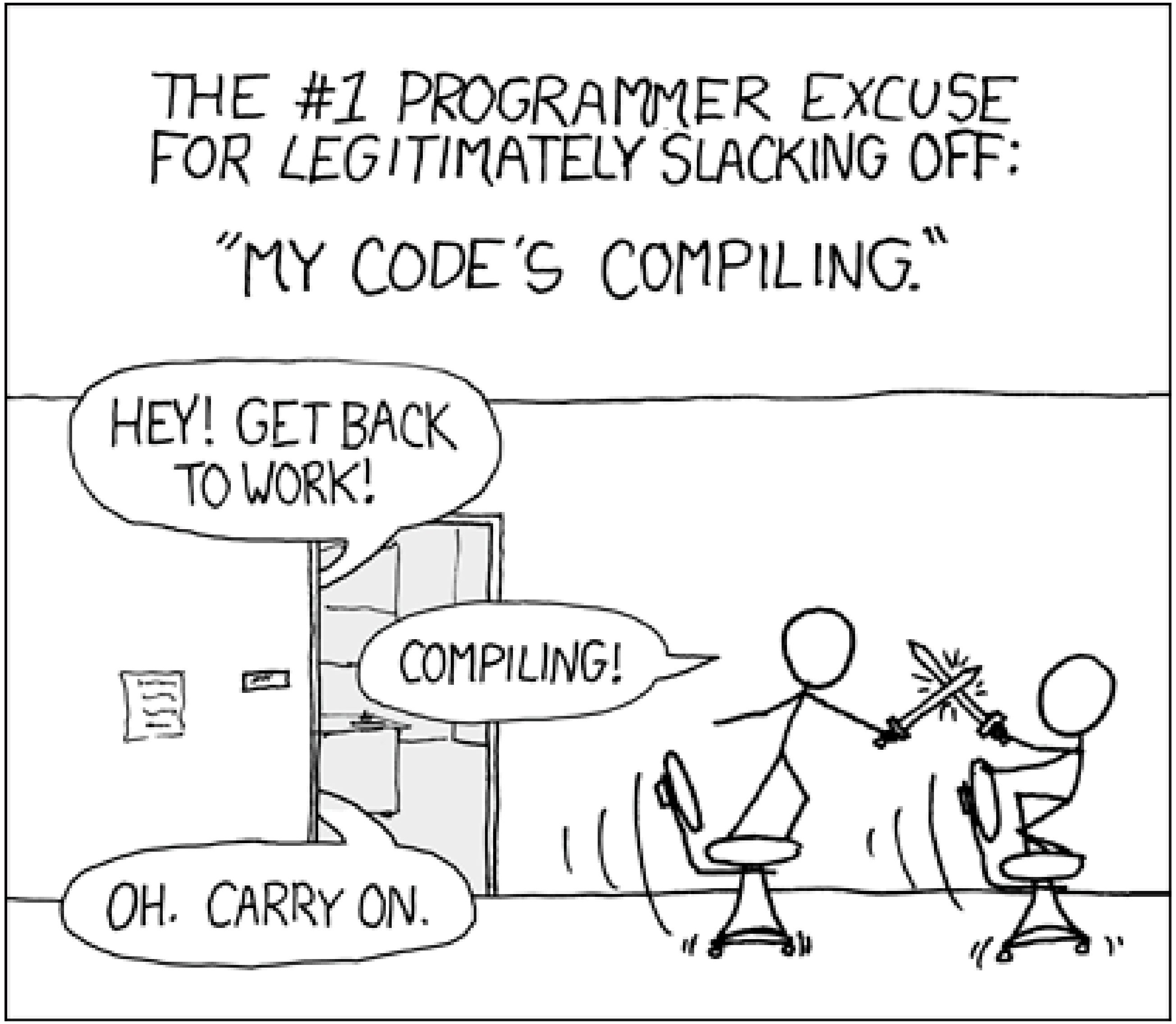
```
#include <stdio.h>
#include "helper.h"

int n = 5;

int main() {
  int i = sum(n,a);
  int j = inc(i);
  printf("%d+1 = %d\n",i,j);
}
```

- Certain optimizations only executed when given *programmer hints*
- E.g., function inlining
  - Replace a function call by *copying the body of the function* you're calling into your body

helper.h

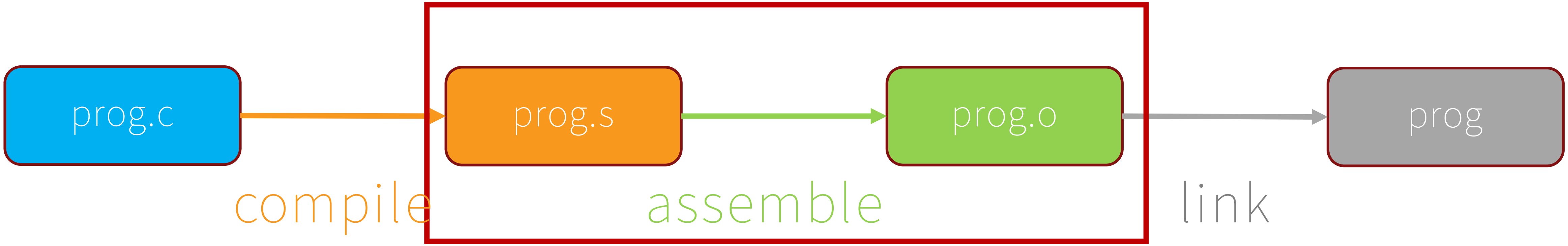
```
extern int a;
int inc(int n);
int sum(int i, int j) {
      return i + j;
};
```

Cornell Bowers CIS
Computer Science



http://xkcd.com/303/

Compiling – From **C** to an **Executable**



- People saying "compile" usually mean: compile + assemble + link
- It's what happens when you run:

```
gcc -o prog prog.c
```

- **Last Step = "Load" program into memory (i.e., running it)**

Assembler

- Input: *.s
  - Program code -- assembly instructions, pseudo-instructions
  - Program data
  - Alignment, memory & type metadata (layout directives)
- Output: *.o
  - "Object File"
  - Operating System-Specific
  - Binary machine code

```
as -o prog.o prog.s
```

```
addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15
```

→

```
00100000000010100000000000001010
00000000000001010010100001000000
00100000101001010000000000001111
```

Assembler

- Need to translate pseudo-instructions to real instructions
  - LI (“load immediate”) -> LUI + ADDI, or just ADDI
  - MV (“move”) -> ADD
  - Other common translations in RISC-V handbook

```
mv x4, x2
```

↓

```
add x4, x2, x0 #but in binary
```

- Symbols & References
  - Similar information to the Assembly file
  - Global labels – externally “exported” symbols (global variables, exported functions)
  - Local labels – only used within the object file
    - Present as metadata, but also removed from assembly instructions

```
    bne x1, x2, L1
…
L1: add x2, x2, x4
```

↓

```
bne x1, x2, 12
…
add x2, x2, x4
```

What’s in an Object File -- Binary Format

- Header
  - Formatting information, size and position of segments
- Text Segment
  - Instructions
- Data Segment
  - Constants / other static data
- Debugging Information
  - Line number / variable name -> instruction / memory mapping
- Symbol Table
  - Global and Local References

If you ever *do need to read an object file*

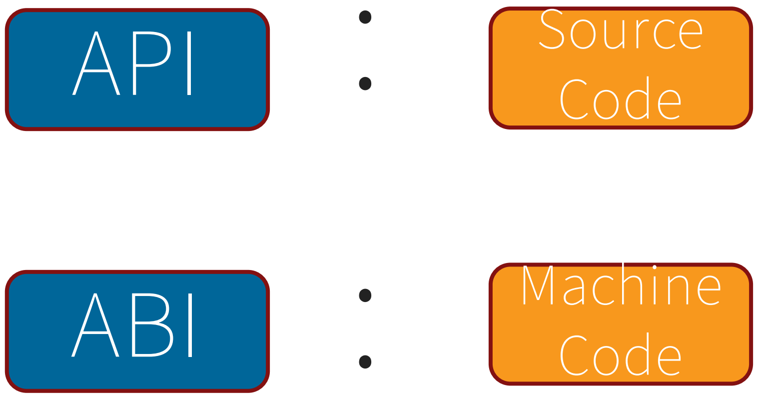
```
objdump -D prog.o
```

- Probably only need to do this when working on embedded systems or when you're writing "inline assembly" or buffer overflow assignment

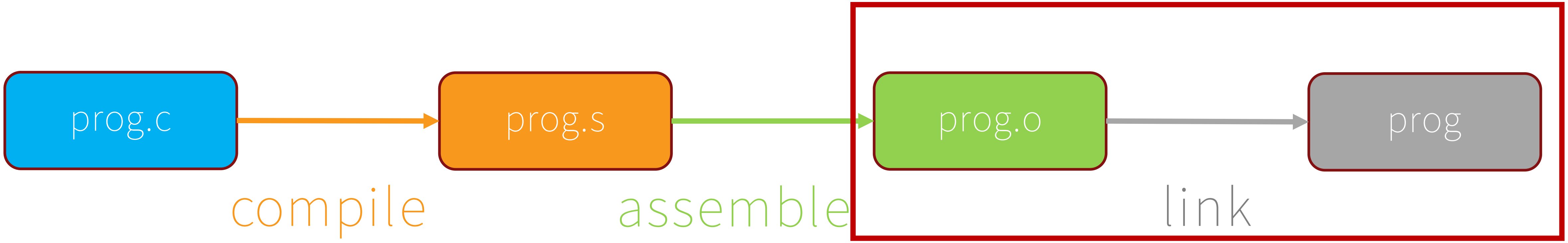
```
prog.o: file format elf32-littleriscv

Disassembly of section .text:

00000000 <main>:
    0: fe010113     addi sp,sp,-32
    4: 00113c23     sd   ra, 24(sp)
    …
Disassembly of section .sdata:
    …
```

Application Binary Interface (ABI)

- Specific to Operating System
  - Describes how to load the program into memory
  - Describes how to run the code
  - Describes which functions/variables it exports
- Unix:
  - Executable and Linkable Format (ELF)
  - Common Object File Format (COFF)
- Windows:
  - Portable Executable (PE)
- Mac:
  - Mach-O



Compiling – From **C** to an **Executable**

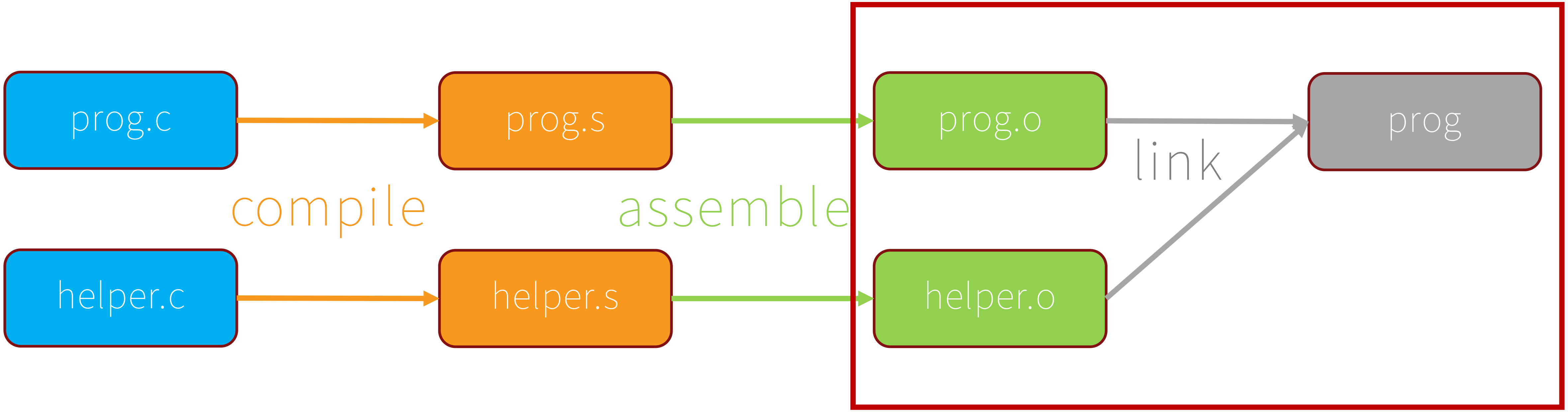


- People saying "compile" usually mean: compile + assemble + link
- It's what happens when you run:

```
gcc -o prog prog.c
```

- Last Step = "Load" program into memory (i.e., running it)

Compiling – From **C** to an **Executable**



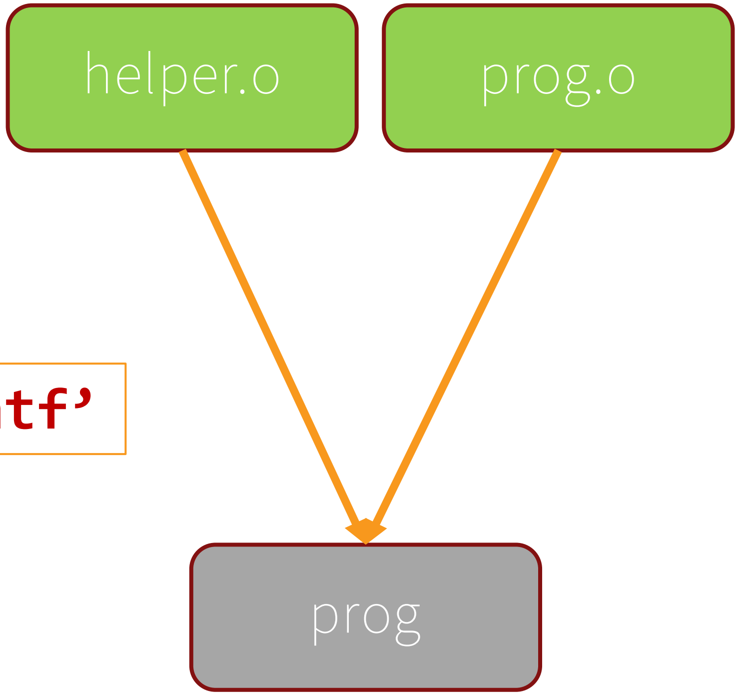
- Separate compilation units (files)
  - Change in only 1 file? Only recompile 1 file (unless this changed an *interface – e.g., the argument types to a function*)
- Link object files together into a *single* executable

Linker

- Combine object files into an executable
- Time to resolve all symbols!

```
ld -o prog prog.o helper.o
```

```
prog.c:(.text+0x44): undefined reference to `printf'
```



Pro Tip!

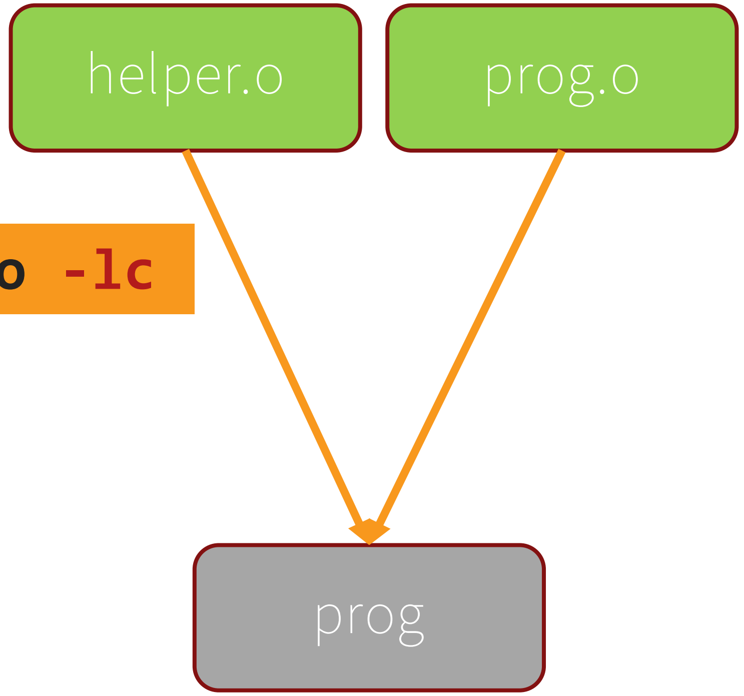
Your program needs to be linked against the "standard library" if using 'built-ins'

```
ld -o prog prog.o helper.o -lc
```

Linker

- Combine object files into an executable
- Time to resolve all symbols!

```
ld -o prog prog.o helper.o -lc
```

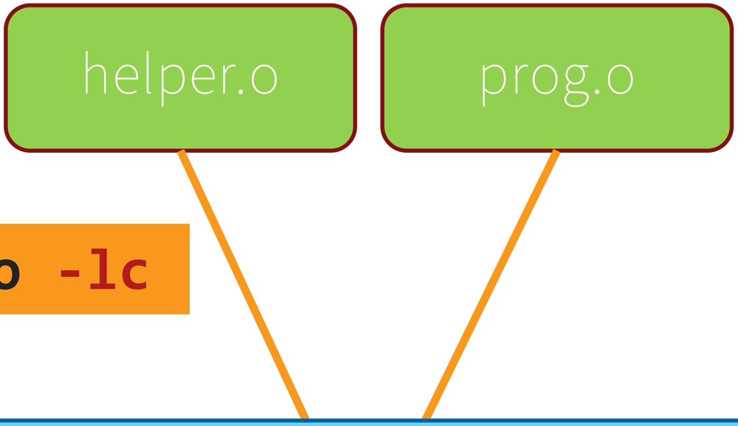


- Each object file "imagines" it has its own main memory array (a.k.a. *address space*)
  - Linking relocates code & data
  - Merge text & data sections
  - Replace final set of labels with offsets
- Record top-level entry point ("main")
- Format still OS specific (conform to the ABI)

Linker

- Combine object files into an executable
- Time to resolve all symbols!

```
ld -o prog prog.o helper.o -lc
```



- Each object file "imagines" it has its own main memory array (a.k.a. *address space*)
  - Linking relocates code & data
  - Merge text & data sections
  - Replace final set of labels with offsets
- Record top-level entry point ("main")
- Format still OS specific (conform to the ABI)

Why is the RISC-V JAL instruction defined to be PC-Relative?

Mnemonic	Description
JAL rd, **offset**	R[rd] = PC+4; PC=PC + **imm** << 1

Position-Independent Code

- If you can't move code or data around, it's difficult to link against it

- When instructions use *PC-Relative* addresses, it's much easier to move code & data around.

Why are RISC-V instructions defined to be PC-Relative?

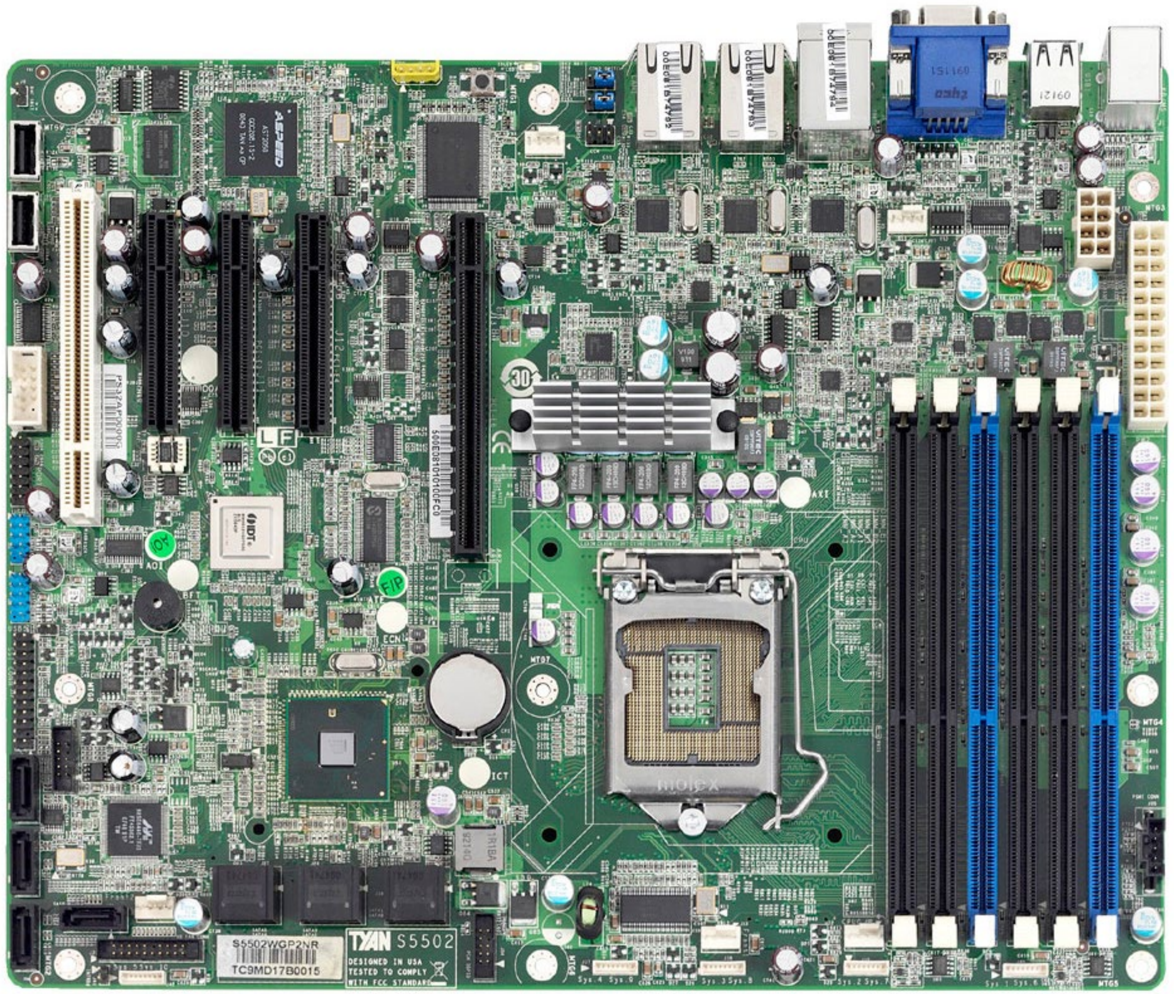
Mnemonic	**Description**
JAL rd, **offset**	R[rd] = PC+4; PC=PC + **imm** << 1
AUIPC rd, **offset**	R[rd] = PC + **imm** << 12;

Static Libraries

- A collection of object files (also called an *archive*)
- Can make your own (e.g., put helper.o in an archive)
- Only link the objects *we need* in our executable
- A bunch of standard ones come with your OS (e.g., libc)
    - Typically each object file ~ one function or one family of functions
    - printf.o, read.o, exit.o, rand.o
    - Specific to OS – systemcall heavy code

System Calls

- ISAs do not have instructions to
  - Write to files
  - Draw on the screen
  - Communicate over the network interface
  - Start a new process
- These are properties of the OS + Peripheral Hardware
- OS + Firmware are responsible for this code (usually involves R/W to special memory addresses)
- syscalls are ISA instructions that transfer control to OS so it can execute these functions

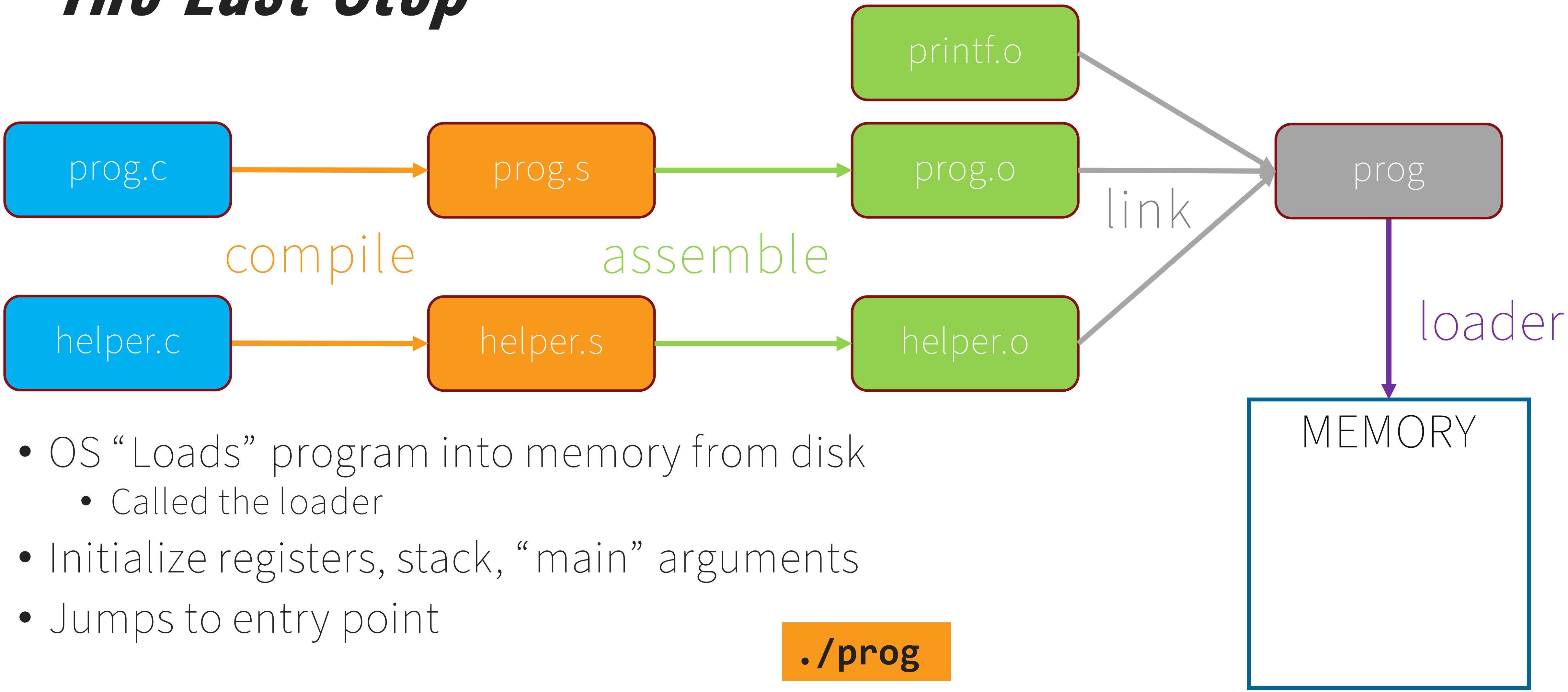


SandyBridge Motherboard, 2011
http://news.softpedia.com

Take CS 4410 for a boatload more info!

Compiling – From **C** to an **Executable**

The Last Step



- OS "Loads" program into memory from disk
  - Called the loader
- Initialize registers, stack, "main" arguments
- Jumps to entry point

```
./prog
```

```
rv qemu prog
```

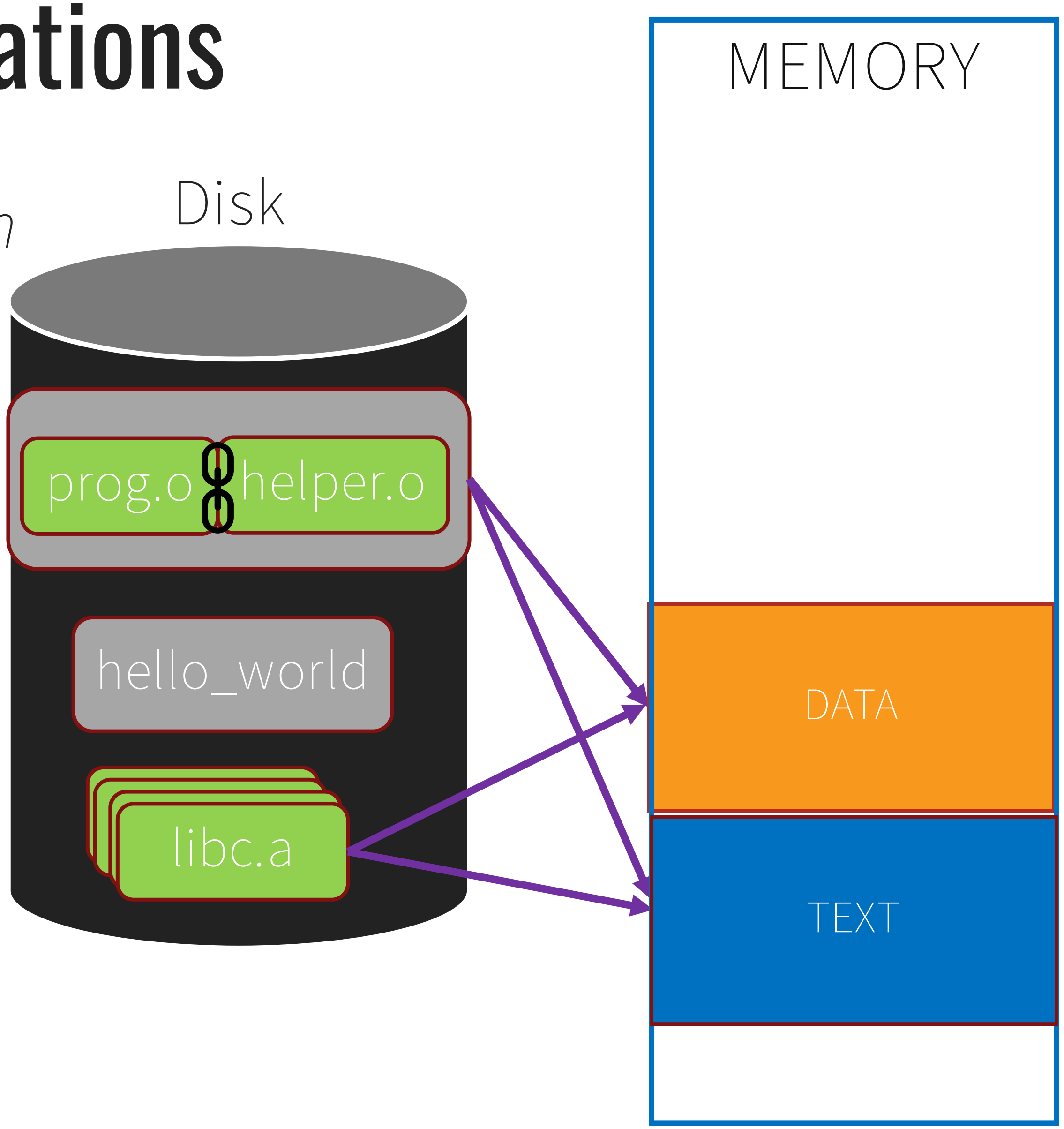
Questions?

- We're touching on Operating Systems, Compilers, and Memory Layout details – so we're eliding a lot of the subtlety

- Are there details about this process you're curious about?

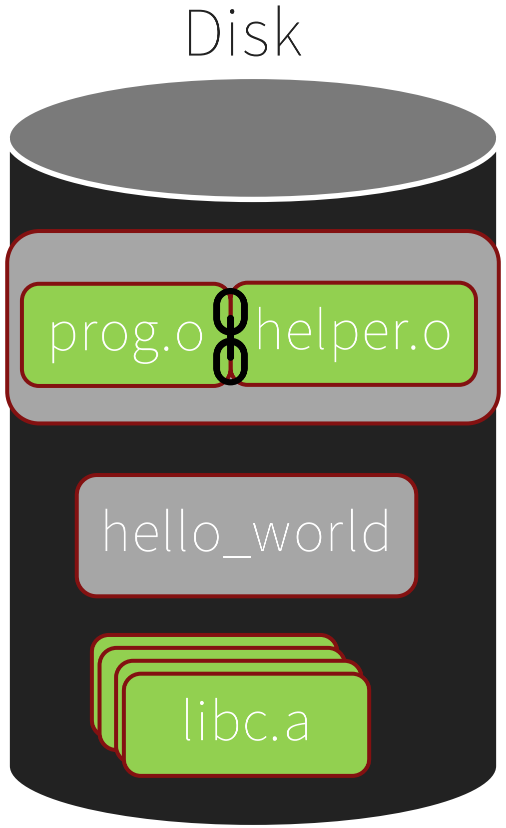
Shared Libraries – Optimizations

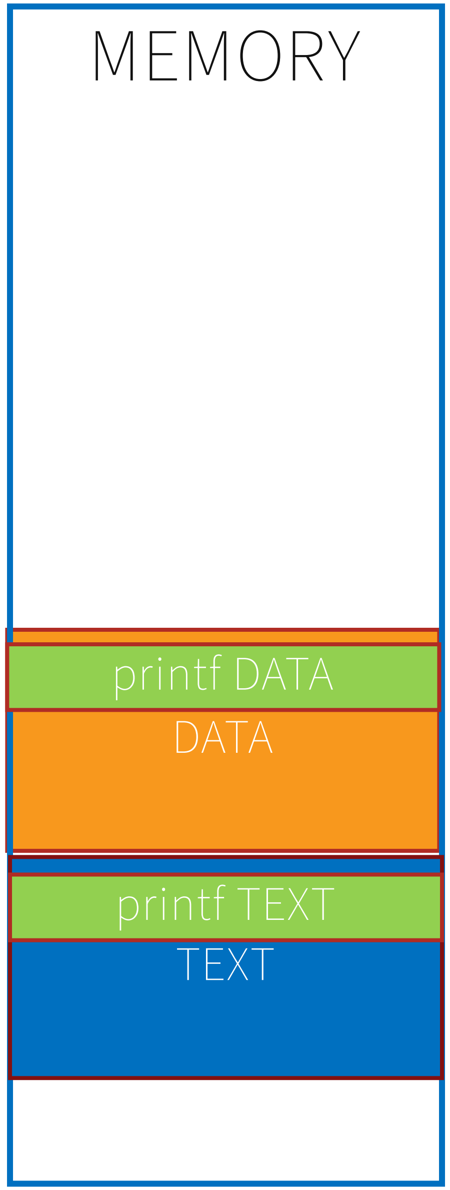
- libc is used *by almost every program*
  - Don't want *copies* in *every executable*
  - Can assume everyone uses it (common case!)
- Static Loading:
  - Loader does the linking right before starting the program
    - Only 1 copy of shared library on disk
    - Can *update* or *customize* library without re-linking



Shared Libraries – Optimizations

- libc is used *by almost every program*
  - Don't want *copies* in *every executable*
  - Can assume everyone uses
- Static Loading:
  - Loader does the linking right before starting the program
    - Only 1 copy of shared library on disk
    - Can *update* or *customize* library without re-linking
  - Can pick fixed parts of the address space to store their code & data – no matter who calls them! (no relocation necessary)
- These files look like: `libgcc.so`





Another Linking Option

- Static Linking
- Big Executables (and TEXT segment)
- Some loading cost (for shared libs)
- Fewer (usually no) compatibility problems
- No runtime or load-time updates

- Dynamic Linking
  - Use **Virtual Memory** to link code at runtime
- Small executable (and TEXT segment if code not called)
- Very little load time – some runtime cost
- Potential compatibility issues (not discovered until runtime)
- Can *dynamically* update code

Another Linking Option

- Static Linking
- Big Executables (and TEXT segment)
- Some loading cost (for shared libs)
- Fewer (usu problems
- No runtim updates

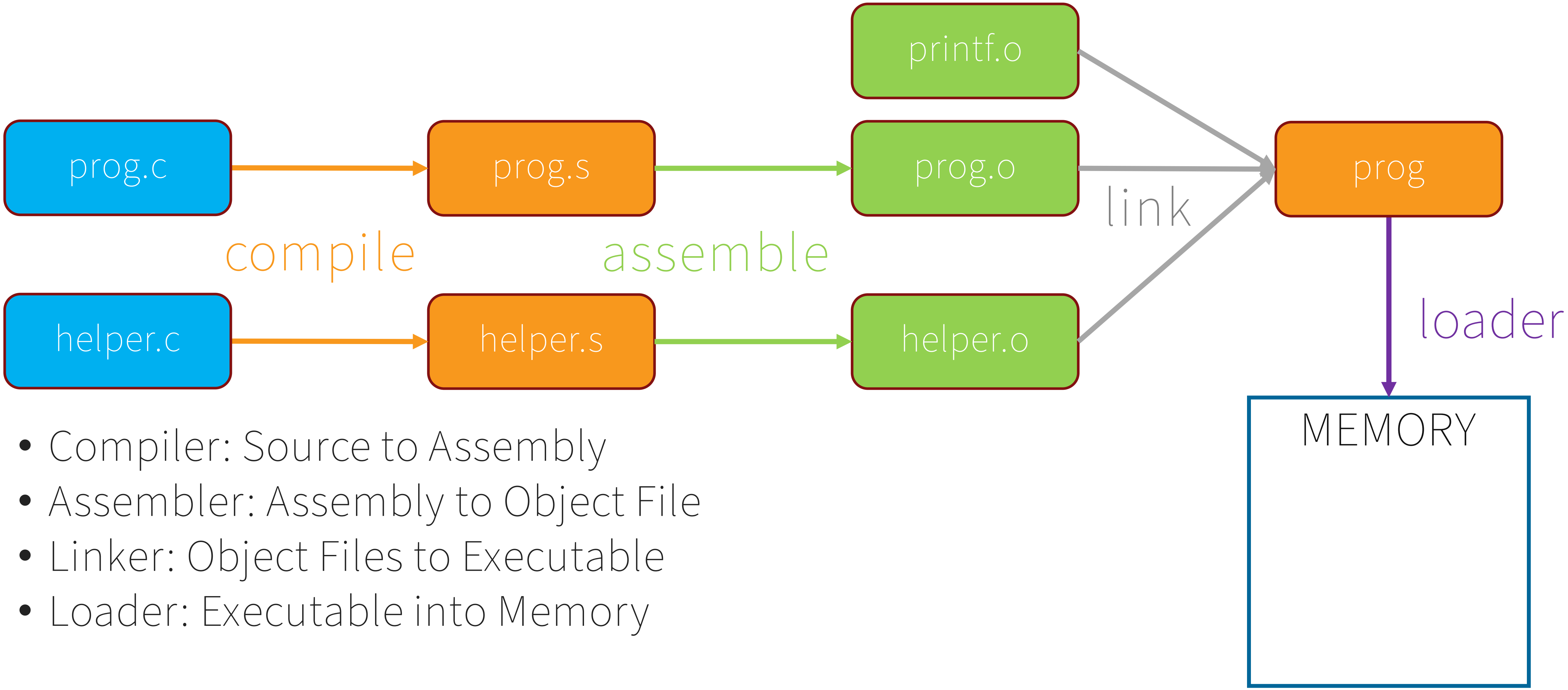
> We'll talk about VIRTUAL MEMORY in a few weeks.
> It provides the *illusion* that every program gets ALL THE MEMORY!!!

- Dynamic Linking
  - Use **Virtual Memory** to link code at runtime
- Small executable (and TEXT segment if code not called)
- Very little load time – some runtime cost
- Potential compatibility issues (not discovered until runtime)
- Can *dynamically* update code

Why The Gory Detail?

- Goal for the Course
  - Understand, from "top to bottom" what happens when your program runs on a computer
- Debug errors you *will see* as a programmer
  - Building low level code
  - Making builds "portable"
- Efficiency of Builds
  - What tradeoffs you can make while compiling/linking to save on:
    - Space, compilation time, program efficiency

Compiling – From **C** to an **Executable**



- Compiler: Source to Assembly
- Assembler: Assembly to Object File
- Linker: Object Files to Executable
- Loader: Executable into Memory