How to extend service, locally?

Yifan Wang

Table of Content

Logistics

- Skill Workshop: Rahul and Zack's TA hour
- Partner finding social event, RSVP: https://edstem.org/us/courses/26354/discussion/1711745
- Slide deck, recording

Compiling

What does compiler do?

- Step 1: Tokenization

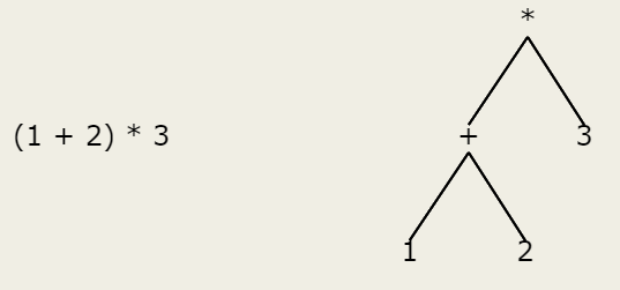
  if (b == 0) a = "hi“ => |if|(|b|==|0|)|a|=|"hi"|

  - Separate operands and operator
  - Operators are written in lexing rules, often in the form of regular expressions.
  - The result is the form made up of tokens.

What does compiler do?

- Step 2: Syntactic Analysis

Token stream -> AST (Abstract Syntax Tree),



What does compiler do?

- Step 3: Type Checking

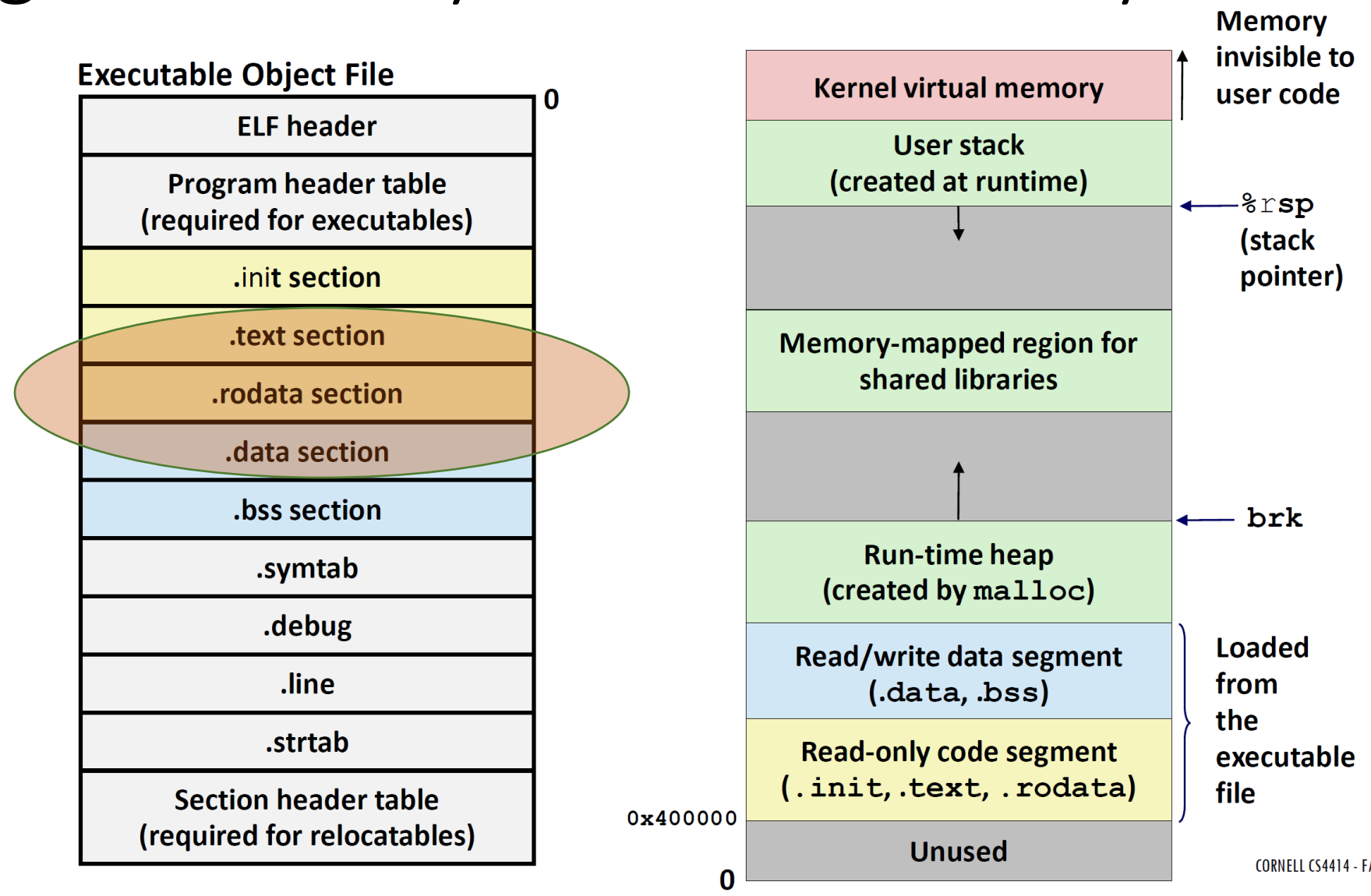
Types + Inference Rules

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \ (\textsc{Plus})$$

What does compiler do?

- Step 4: Translation
  - Translate AST to lower language, which is close to the assembly language. Another layer of abstraction.
    - Assembly language can be OS dependent: ATT (Linux) movl $1, %eax vs. Intel (Win) movl eax, 1.
    - Assembly language can be updated with new CPUs, richer choice of instructions.
  - Choosing the right instruction has a huge impact on the performance of the compiled program.
  - Then, lower language -> assembly language, this step can be optimized for a wiser choice of instructions.
  - Finally, assembly language -> machine code. Hardware dependent, mechanical translation.

Program in filesystem and memory



But now we have a problem

- Do we need to reinvent the wheel?
- Do we need to modify the entire code for a small change like changing a variable name?

Linking

What is gcc?

- A “driver” of tools you’ll use for compiling your program.
- Multiple compilers are included: C, C++, Objective C, Go, and etc.
- Linker is included and invoked in a hidden manner, with the –l parameter.
  - ld is the command to invoke linker alone.

- Support optimization, debugging mode, and etc…

What is a linker?

- A linker takes a collection of object files and combines them into a single object file. But this object file will still depend on libraries.
- Next it cross -references this single object file against libraries, resolving any references to methods or constants in those libraries.
- If everything needed has been found, it outputs the executable files.

Object Files?

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

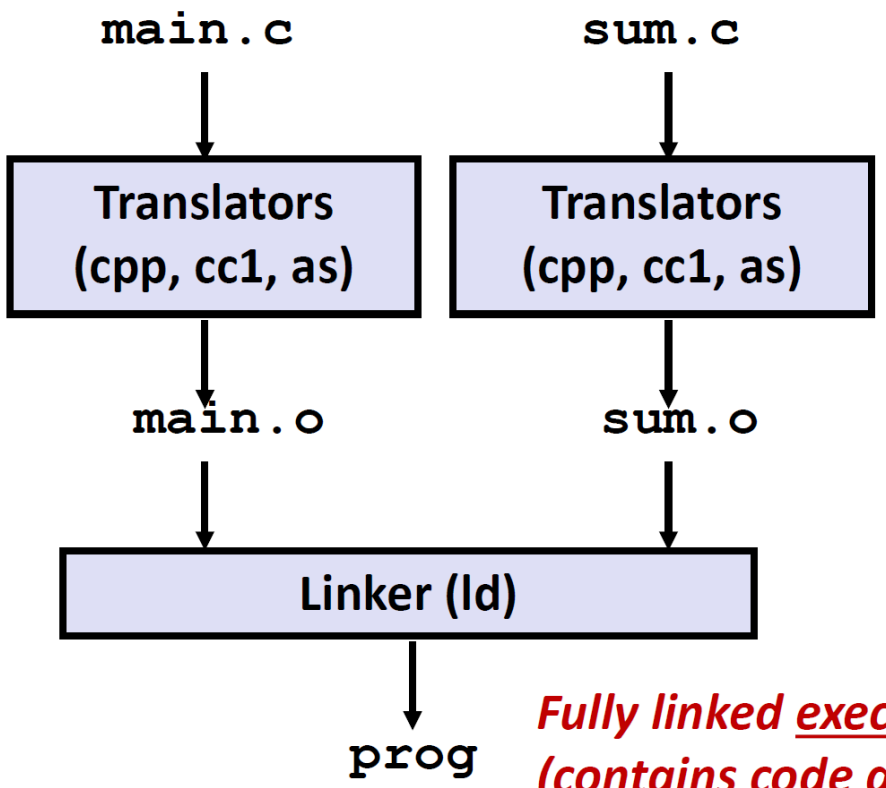
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Object Files?



Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in `main.c` *and* `sum.c`*)*

Object Files?

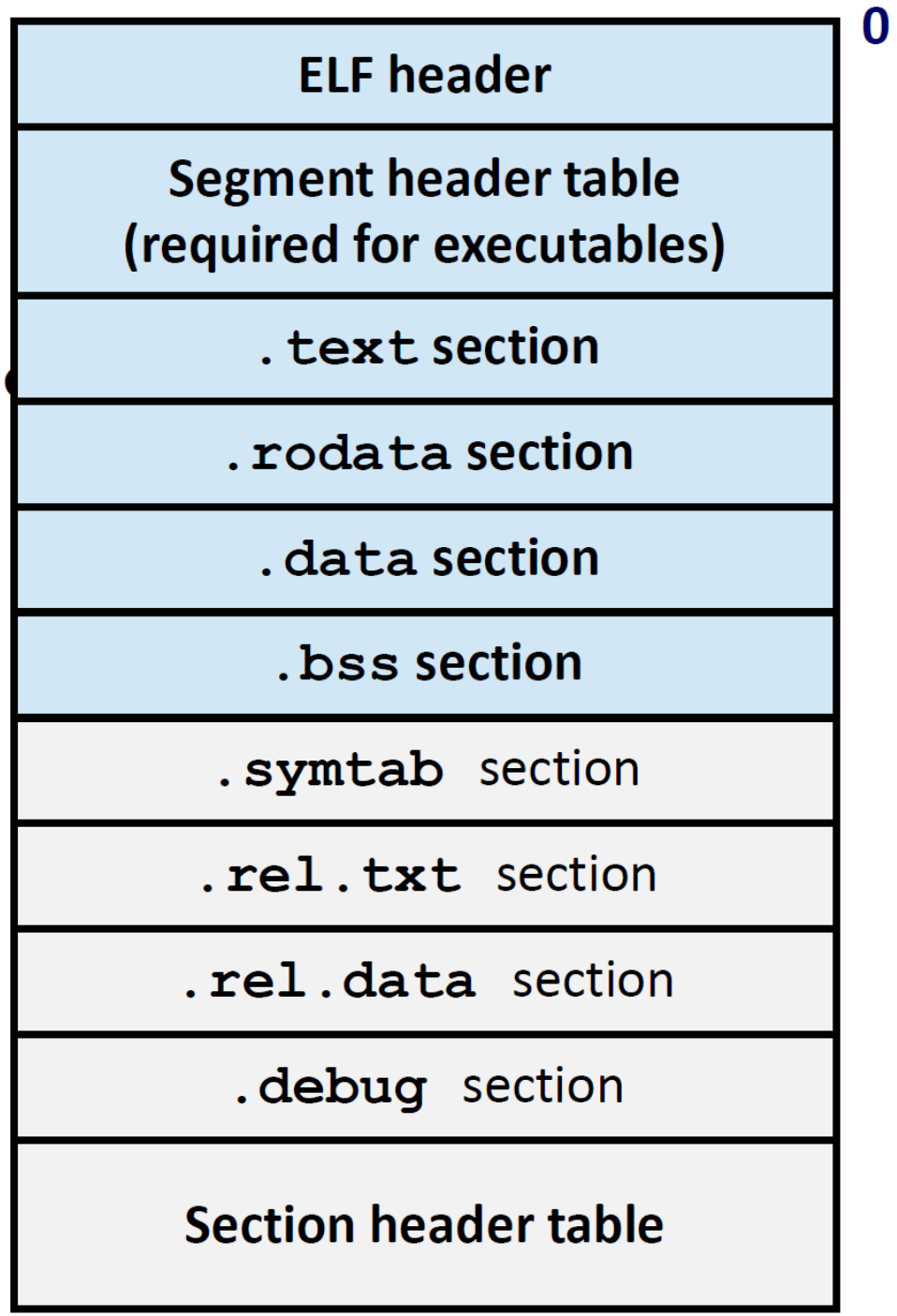
An object file contains “incomplete” machine instructions, with locations that may still need to be filled in:

- Addresses of methods defined in other object files, or libraries
- Addresses of data and bss segments, in memory

After linking, all the “resolved” addresses will have been inserted at those previously unresolved locations in the object file.

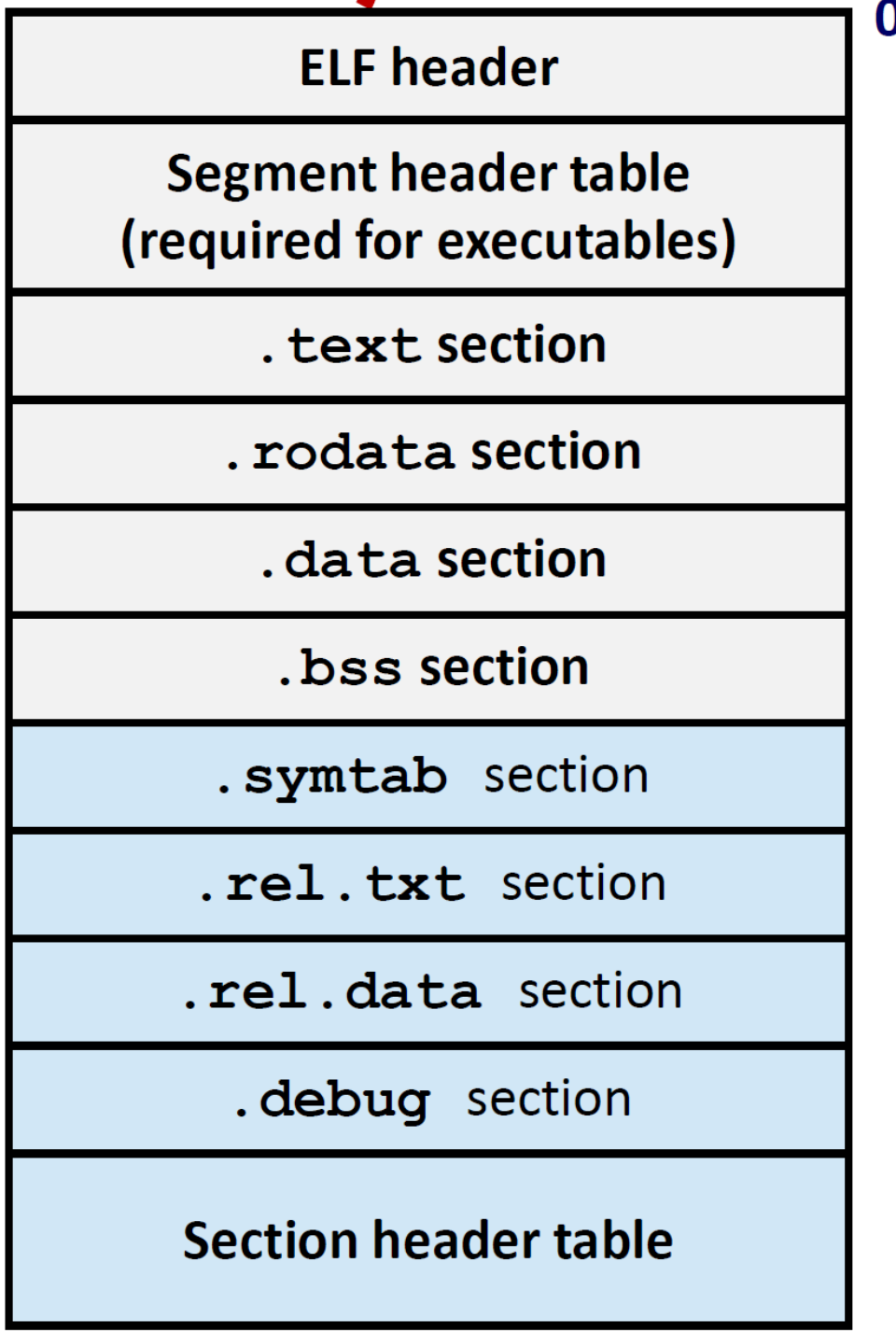
Output

- ELF = Executable and Linkable Format
- Elf header: Word size, byte ordering, file type (.o, exec, .so), etc.
- Segmentation header: page size, virtual memory segmentation size.
- .text code
- .rodata readonly data, jump offsets, strings
- .data initialized global variables
- .bss non-initialized global variables



Output

- .symtab symbol table, static variable name, etc.
- .rel.text relocation info for .text
- .rel.data relocation infor for .data
- .debug debug info (gcc –g)
- Section header table offsets and size of each section



Variables

- Global variable, static variable are on rodata/data
- How about local variables?

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc,char **argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Referencing a global…

…that's defined here

Defining a global

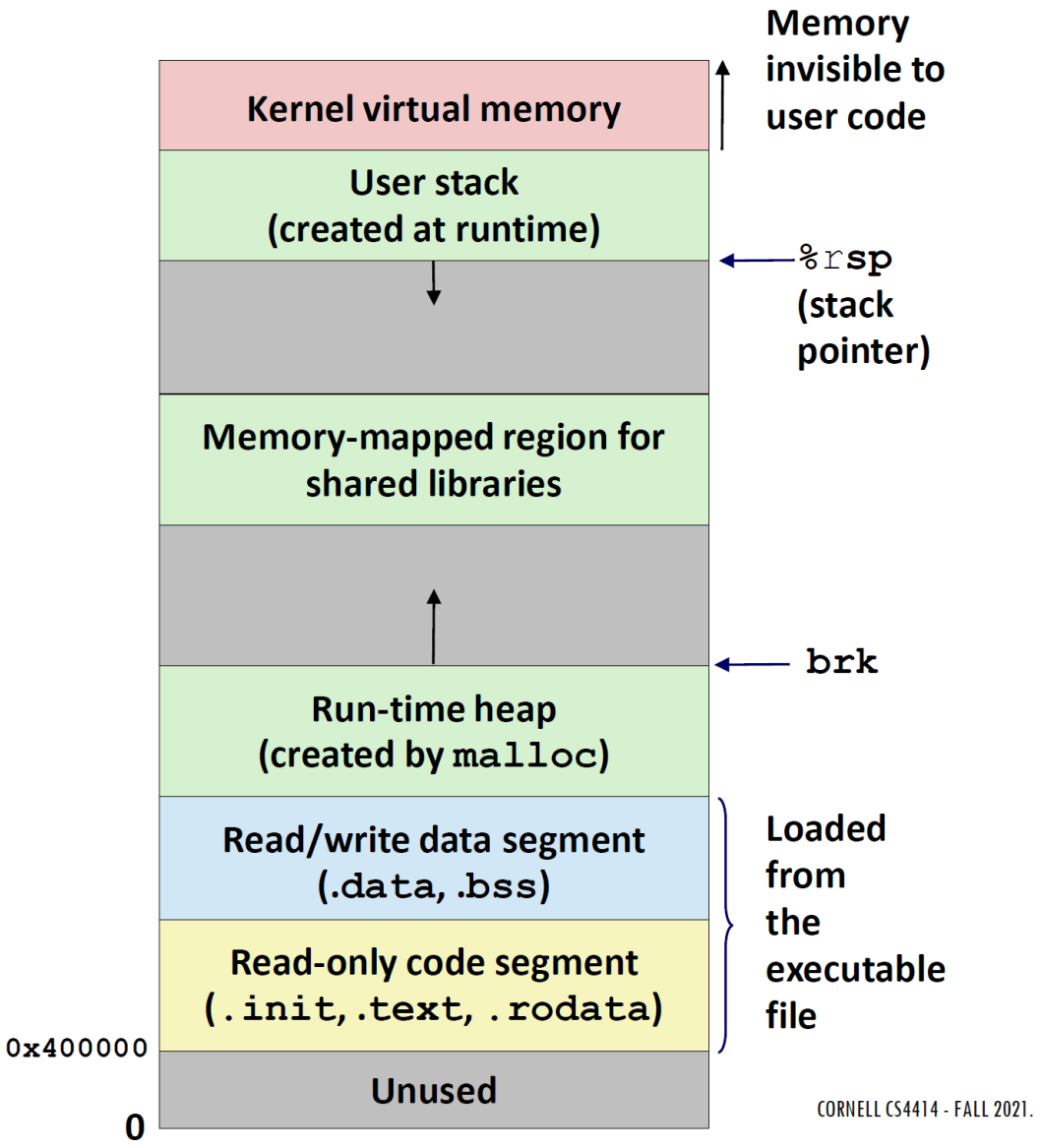
Linker knows nothing of `val`

Referencing a global…

…that's defined here

Linker knows nothing of `i` or `s`

Memory space



Symbols?

Which of the following names will be in the symbol table of `symbols.o`?

symbols.**c**:

```
int incr = 1;
static int foo(int a) {
  int b = a + incr;
  return b;
}

int main(int argc,
         char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

Names:

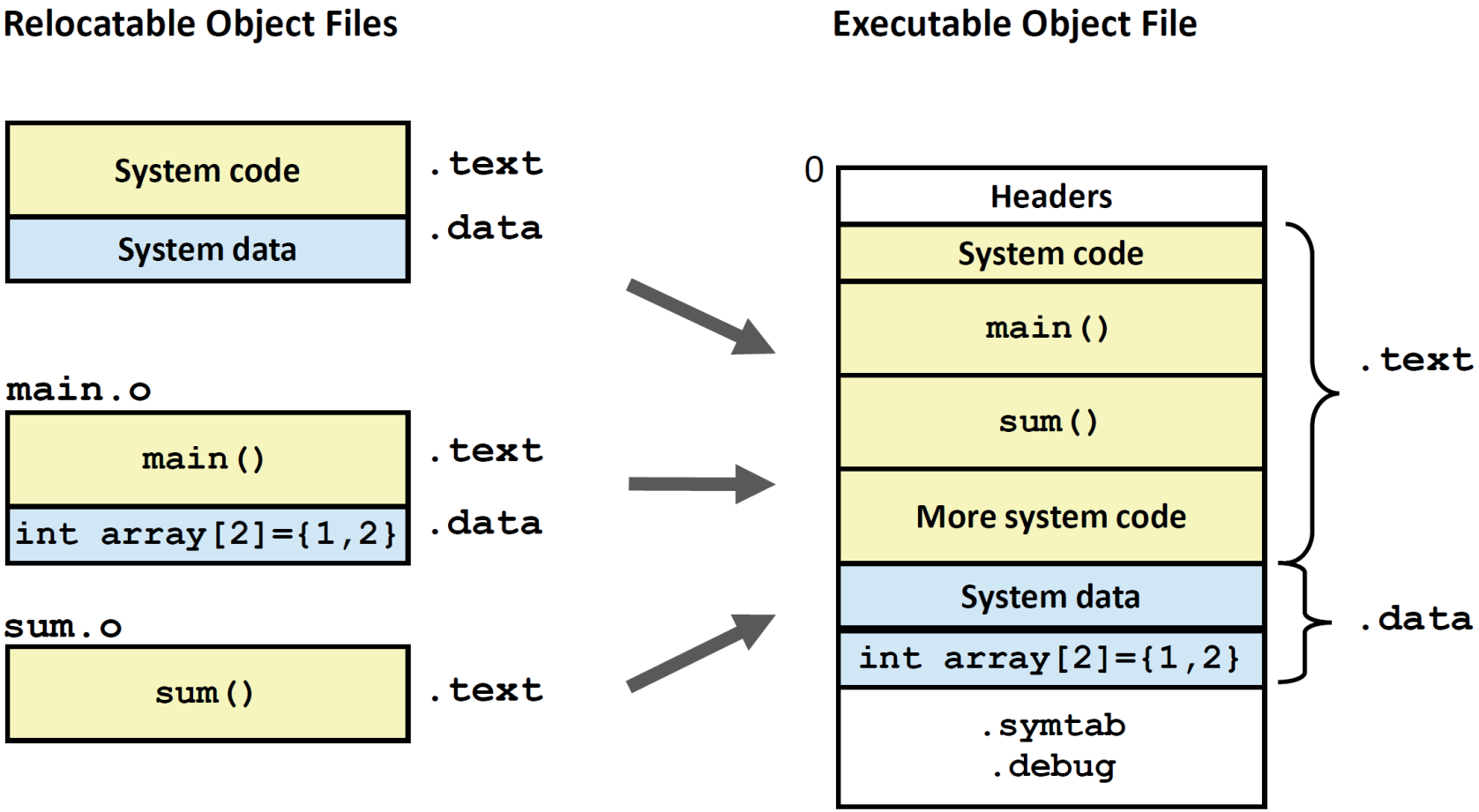
- incr
- foo
- a
- argc
- argv
- b
- main
- printf
- "%d\n"

Can find this with `readelf`:

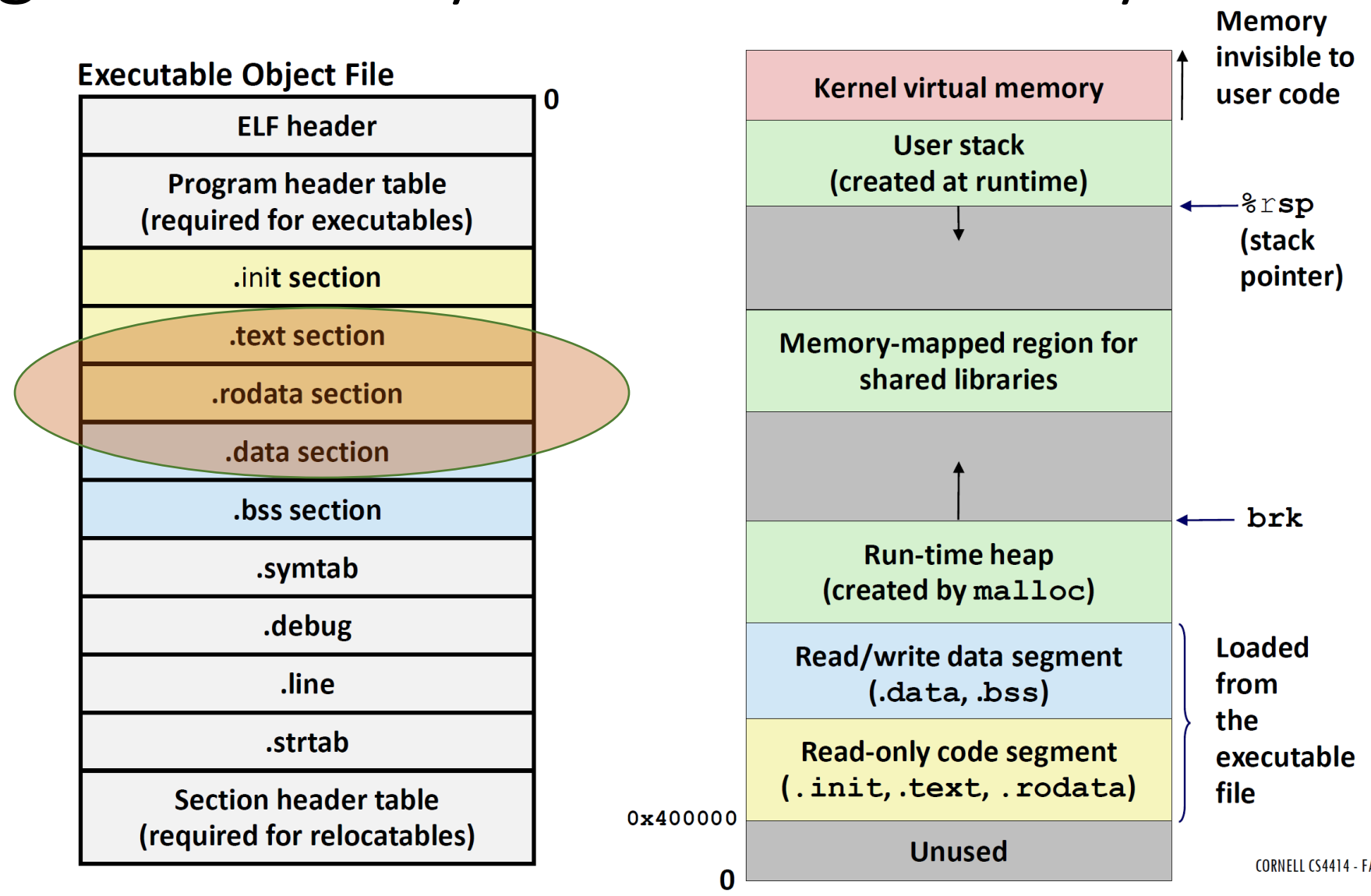
```
linux> readelf -s symbols.o
```

Link Everything Together

- Compiler assigns absolute "location" for code and data.



Program in filesystem and memory



Static Linking

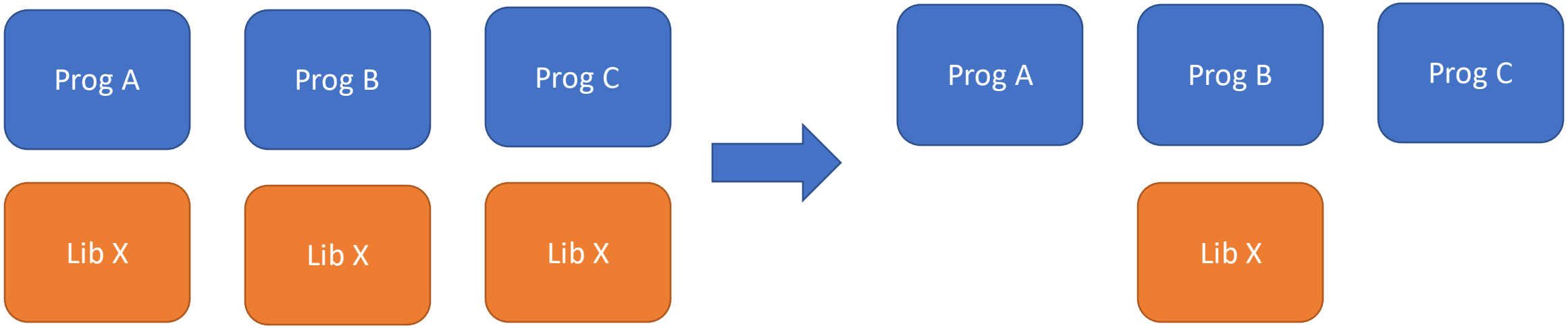
- You can share your code with users.
  - 0-day cutting edge access.
  - Reliability,.
  - Platform dependency, i.e., OS, hardware.
- You can share your library only, i.e., just the .o files.
- At least your code is modularized, with proper namespace settings.

However, I want…

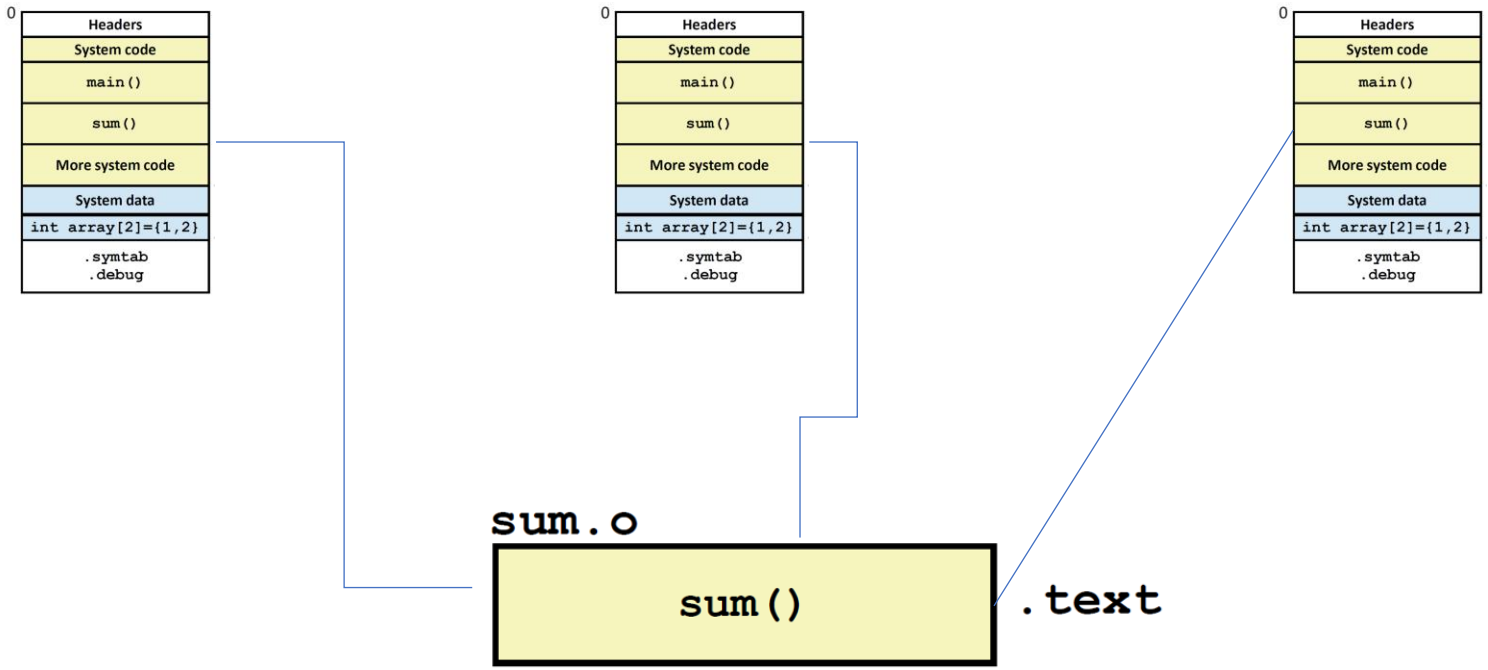
I want to keep a single copy of a library on my computer, not multiple copies of the same .o file for each of the project I created.

I want to keep my executables as is without recompiling, even if the library changes while the access points (API, global/static variable, etc.) remain the same.

Dynamic Linking

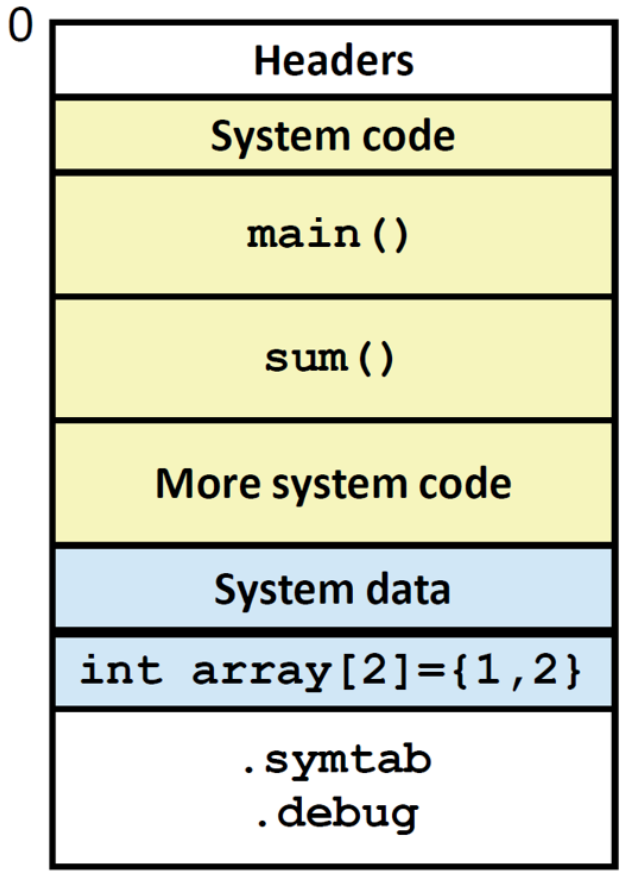


Dynamic Linking



How do we get the correct address?

- Different programs have different main() and hence the location of sum() in .text segment is different between each programs.



Solution: Dynamic Relocation

- We compile the library with –shared –fPIC. This tells the compiler to generate “register offset” addressing
- to put the code segment base address in a specific register (save the old value to the stack!), and the data segment base into a second register (“ “ “). Restore the original values when the method returns.
- –fPIC, all jumps and data accesses in the DLL are “relativized” as offsets with respect to these registers.

Python

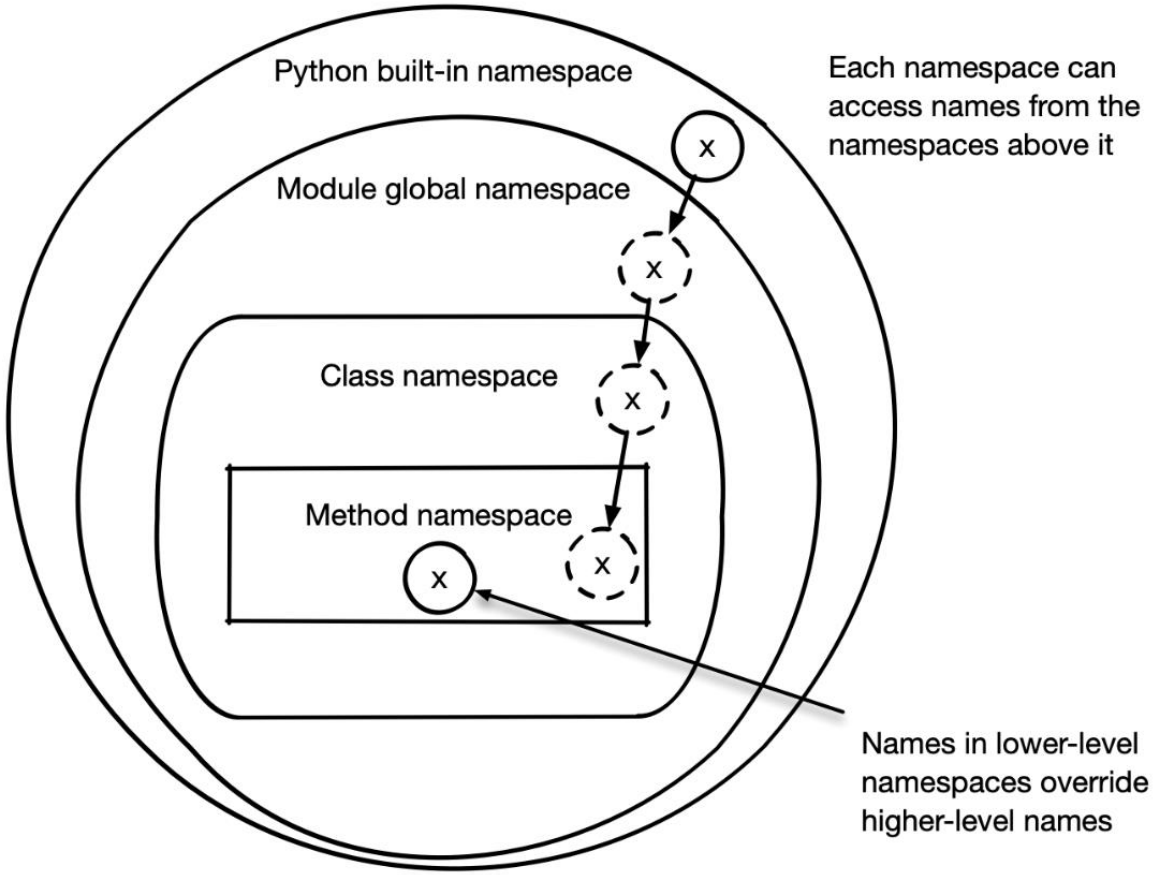
What happens when you import

- Copy the code snippet into your script.
- Build AST, symbol table, etc.
- Translate AST into byte code.
- Run in Python Interpreter and VM.

It is kind of static

- Only one copy of the code on disk. But you still need to keep multiple copies in memory.
- Every time you invoke the program, it is recompiled.

Namespaces


Python built-in namespace
Module global namespace
Class namespace
Method namespace
x
x
x
x
x
Each namespace can access names from the namespaces above it
Names in lower-level namespaces override higher-level names

Namespaces, cont'd

- Sometimes the namespace is hidden
  - print() (built_in namespace)
  - from A import a (add a to the global namespace)
- You can also express explicitly

```
import A
A.a
```