CS35L Software Construction Laboratory

Lab 5: Sneha Shankar

Week 9; Lecture 1

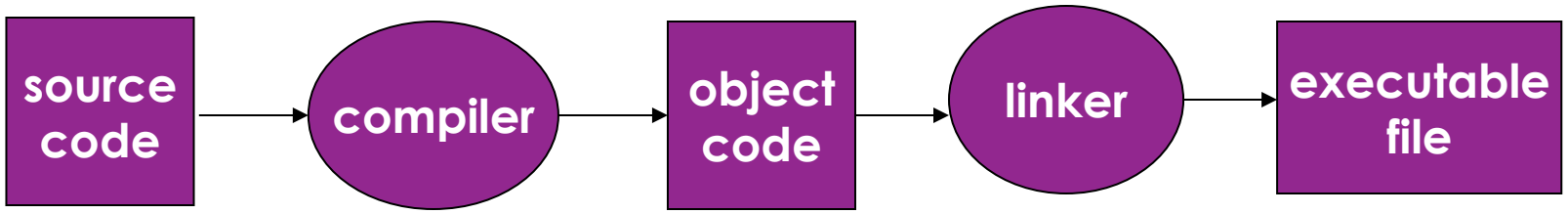
Dynamic Linking

Lifecycle of a program

The following entities help in getting a program to work

- Compiler
- Interpreter
- Assembler
- Linker
- Loader

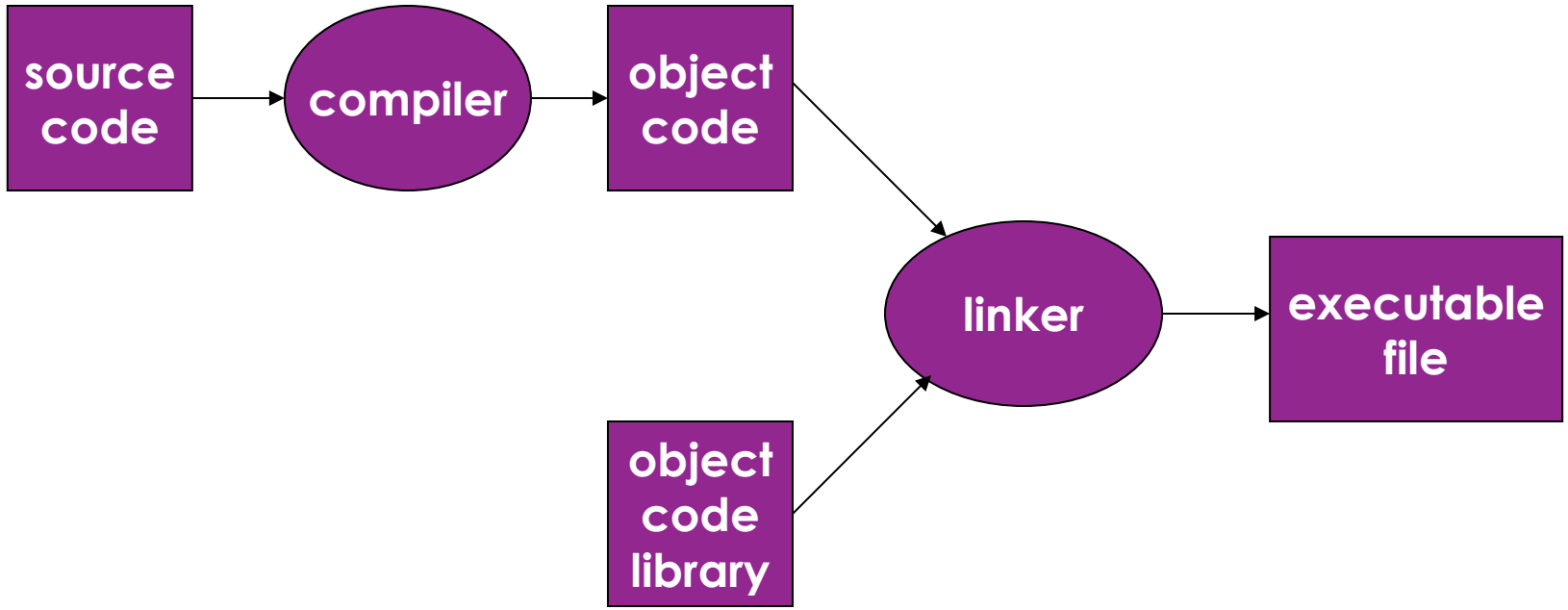
Building an executable file



Translates programming language statements into cpu's machine-language instructions

Adjusts any memory references to fit the Operating Sytem's memory model

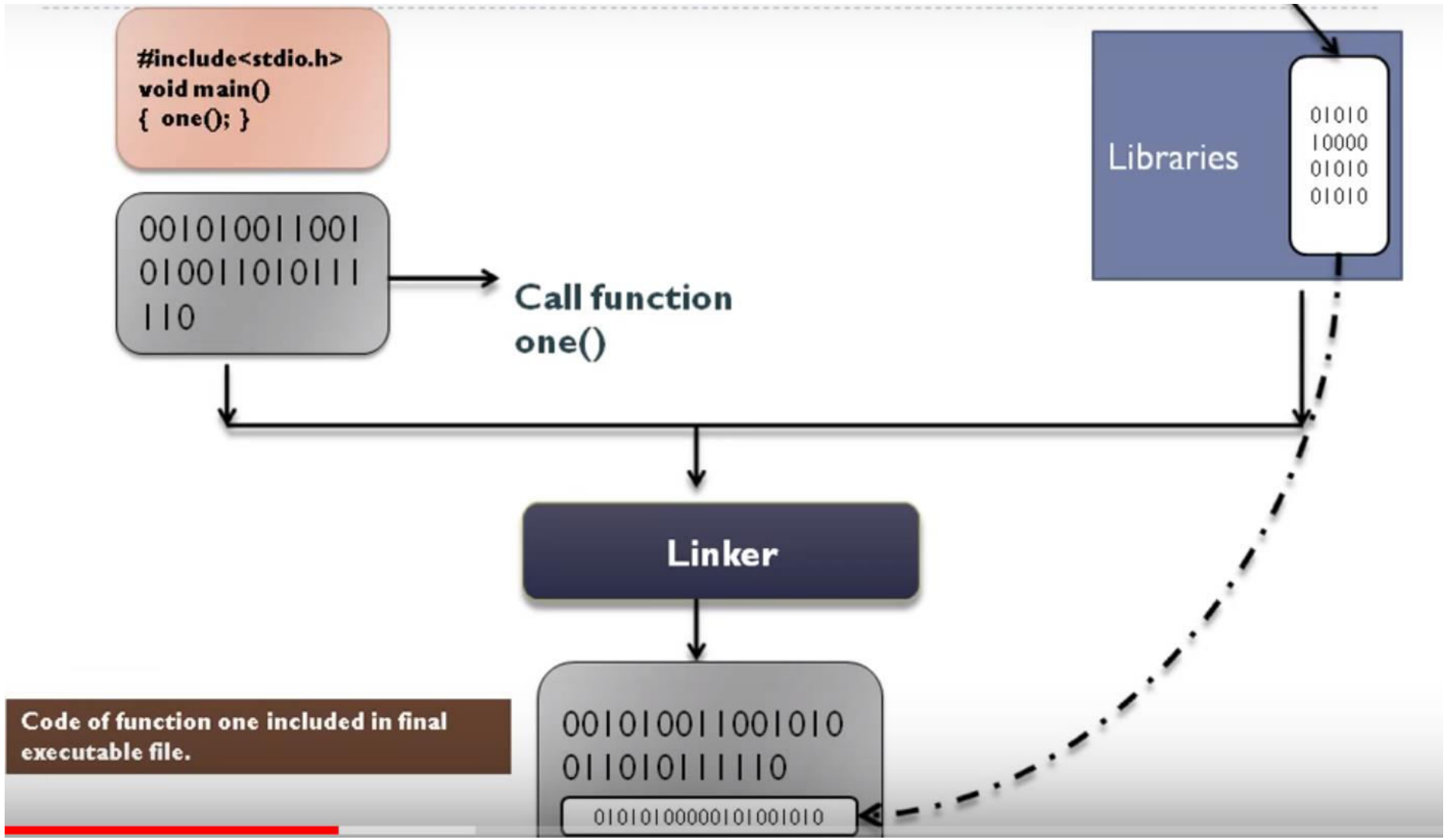
Linking libraries

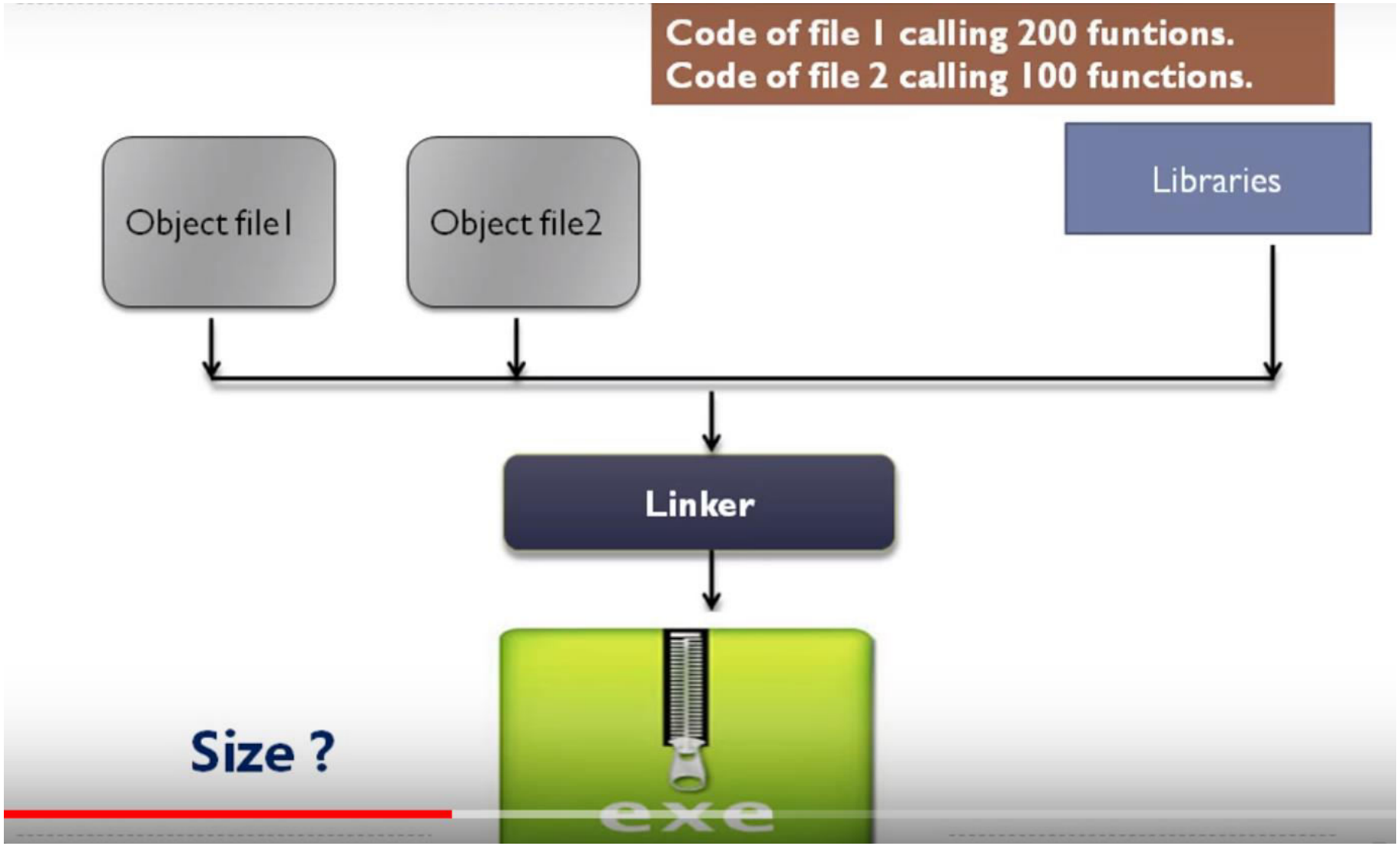


A previously compiled collection of standard program functions

Static Linking

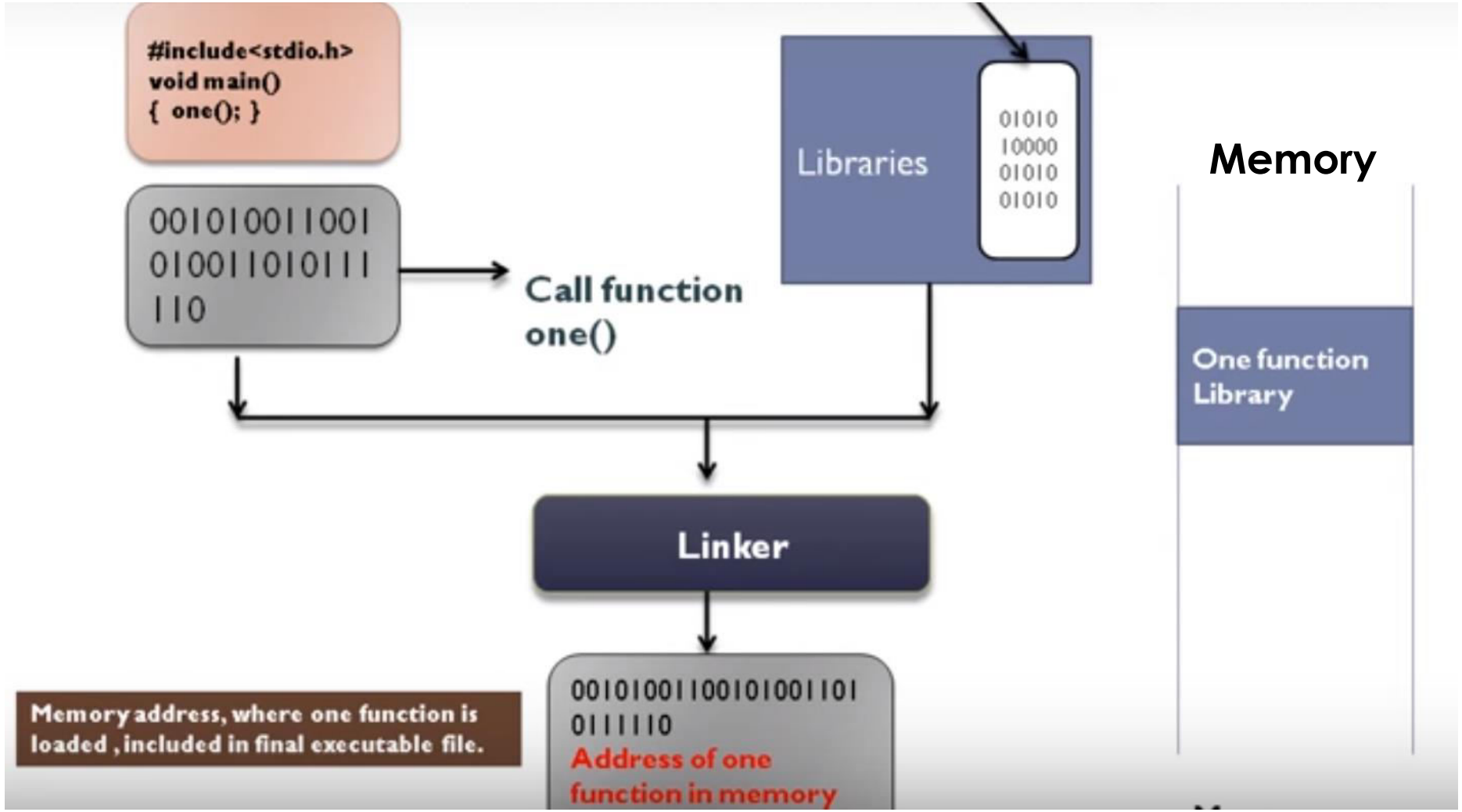
- Carried out only once to produce an executable file
- If static libraries are called, the linker will copy all the modules referenced by the program to the executable
- Static libraries are typically denoted by the .a file extension


#include<stdio.h>
void main()
{ one(); }
00101001100101001101011110
Call function one()
Libraries
01010
10000
01010
01010
Linker
Code of function one included in final executable file.
0010100110010100110101111110
0101010000010100101010

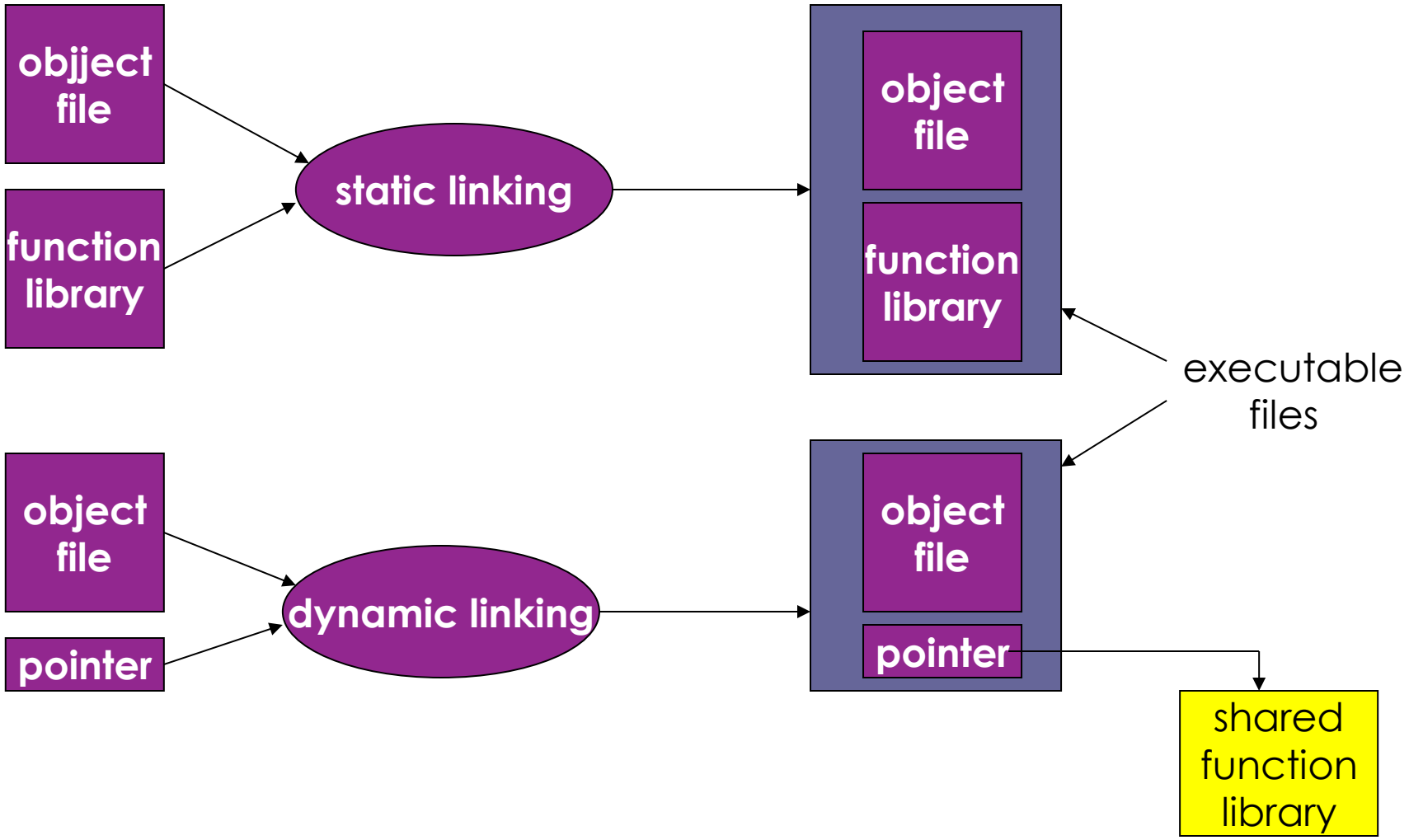

Code of file 1 calling 200 funtions.
Code of file 2 calling 100 functions.
Object file1
Object file2
Libraries
Linker
Size ?
exe

Dynamic Linking

- If shared libraries are called:
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time
- Dynamic libraries are typically denoted by the .so file extension
  - .dll on Windows

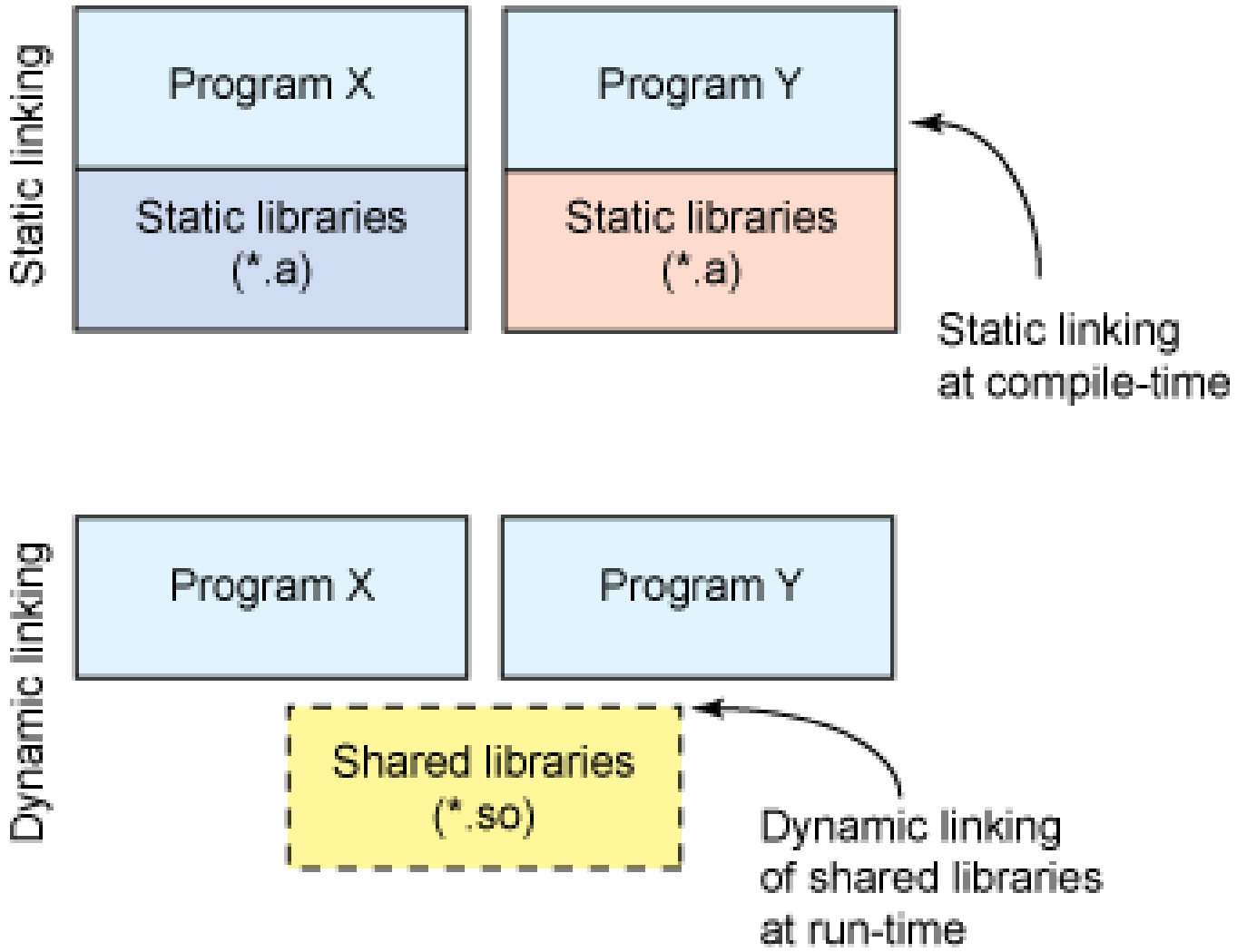

#include<stdio.h>
void main()
{ one(); }
00101001100101001101011110
Call function one()
Libraries
01010
10000
01010
01010
Linker
00101001100101001101011110
Address of one function in memory
Memory address, where one function is loaded , included in final executable file.
Memory
One function Library

Smaller is more efficient



Linking and Loading

- Linker collects procedures and links them together object modules into one executable program
- Why isn't everything written as just one **big** program, saving the necessity of linking?
  - Efficiency: if just one function is changed in a 100K line program, why recompile the whole program? Just recompile the one function and relink.
  - Multiple-language programs


Static linking
Program X
Static libraries
(*.a)
Program Y
Static libraries
(*.a)
Static linking
at compile-time
Dynamic linking
Program X
Program Y
Shared libraries
(*.so)
Dynamic linking
of shared libraries
at run-time

Dynamic linking

- Dynamic vs. static linking resulting size

```
$ gcc -static hello.c -o hello-static
$ gcc hello.c -o hello-dynamic
$ ls -l hello
      80 hello.c
   13724 hello-dynamic
 1688756 hello-static
```

Advantages of dynamic linking

- The executable is typically smaller
- When the library is changed, the code that references it does not usually need to be recompiled
- The executable accesses the .so at run time; therefore, multiple programs can access the same .so at the same time
  - Memory footprint amortized across all programs using the same .so

Disadvantages of dynamic linking

- Performance hit
  - Need to load shared objects (at least once)
  - Need to resolve addresses (once or every time)
- What if the necessary dynamic library is missing?
- What if we have the library, but it is the wrong version?

Learn and Practice

https://www.geeksforgeeks.org/working-with-shared-libraries-set-2/

Lab 8

- Write and build simple "cos(sqrt(3.0))" program in C
  - Use `ldd` to investigate which dynamic libraries your hello world program loads
  - Use `strace` to investigate which system calls your hello world program makes
- Use "`ls /usr/bin | awk 'NR%101==SID%101'`" to find ~25 linux commands to use `ldd` on
  - Record output for each one in your log and investigate any errors you might see
  - From all dynamic libraries you find, create a sorted list
    - Remember to remove the duplicates!