UCLA CS35L

Week 7

Monday

Reminders

- Assignment 6 due this Friday (5/15)
- Assignment 7 due next Friday (5/22)

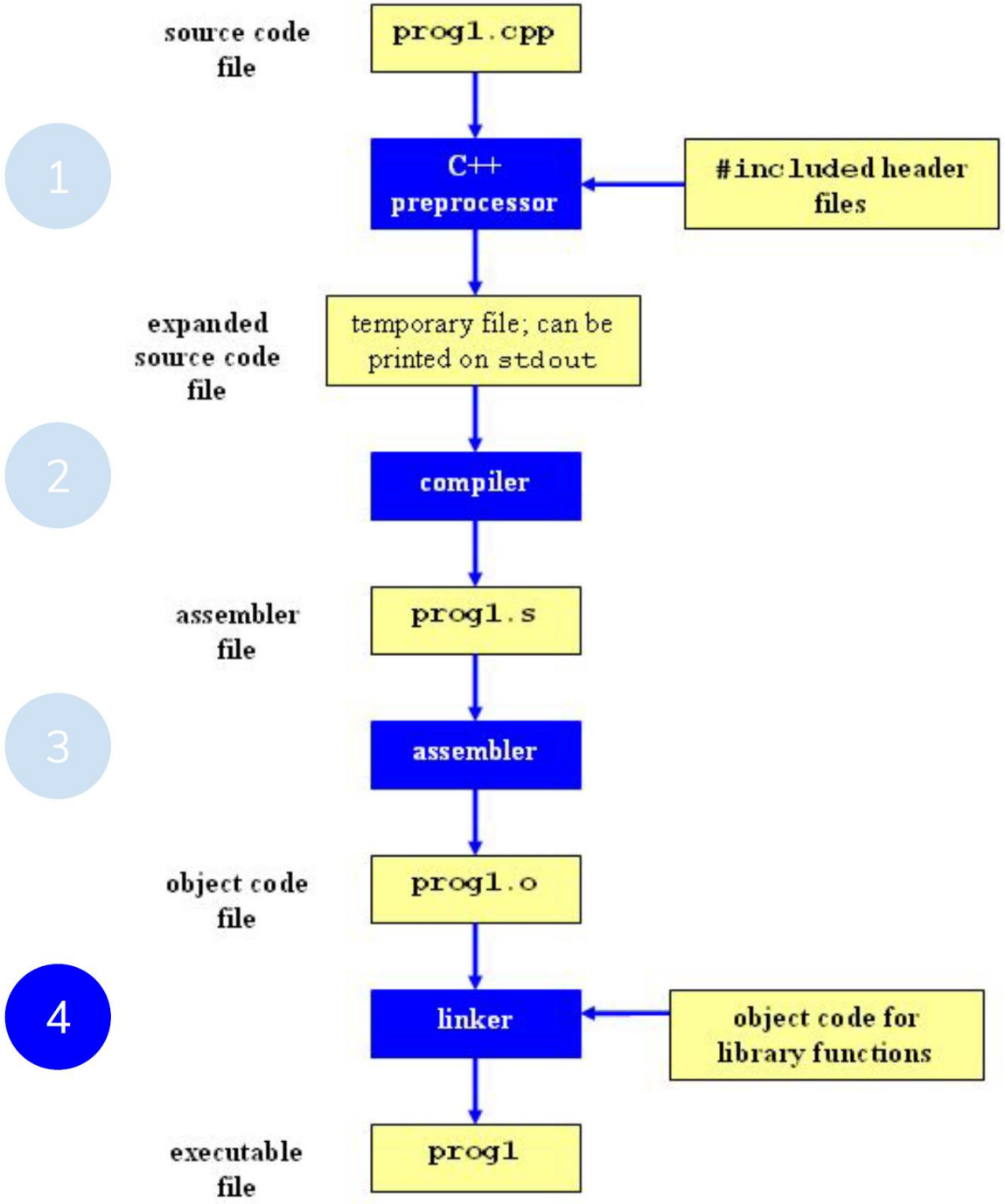
- I randomly assigned groups/timeslots, check the signup sheet
  - I will send out the emails of the class so you can contact your partner
  - First presenters are next Wednesday (5/20) – Ty Koslowski + William Randall
- Reach out to me if:
  - You need to send in a recording due to timezone issues making it hard to present live
  - Your partner does not respond within 2-3 days.

- Anonymous feedback for Daniel
  - https://forms.gle/tZwuMbALe825DBVn8

Linking

What is Linking

- We are interested in Step 4
- Object Code is **linked** with other object code files for library functions, and produce the final executable



What is Linking

- When a program uses external library functions, it includes symbol referencing code defined outside of the current program
- Linking is the process of **symbol resolution** and **relocation**
- Why use linking?
  - Modularity
  - Time efficiency (No need to recompile unmodified source files)
  - Space efficiency (we'll see how dynamic linking helps with this)
- This is done by programs called Linkers

Symbol Resolution

- Programs define and reference symbols (global variables and functions)
- The assembler generates a symbol table which is stored in the object file
- The symbol table is an array of symbol entries, including name and location of symbol
- The linker associates each symbol reference with one symbol definition

Symbol Resolution Example

```
void foo(void);

int main(){
  foo();
  return 0;
}
```

Compiler: Cannot find symbol foo in the current module

Generate symbol table, and entry for foo. Hand off to linker.

Linker: foo is not defined in any input modules that I can find, I can’t resolve it -> error

Relocation

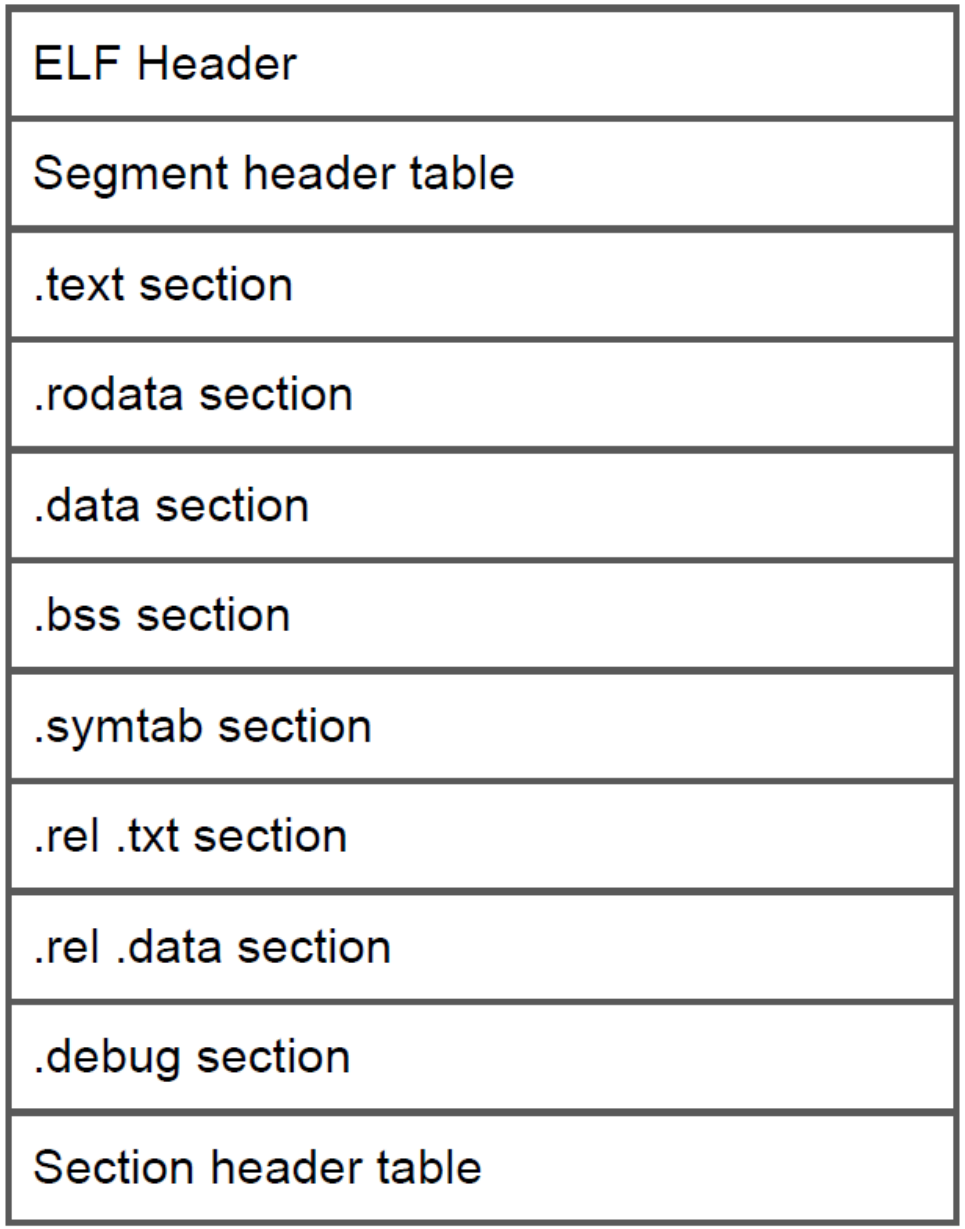
- Now the linker knows the exact size of all the symbols in the file
- Now it can relocate symbols from their relative locations in the .o files to their final memory location in the executable.
  - This needs to be done because the memory location at runtime is different from the memory location at compile time. At compile time we assume all memory starts at 0, but this isn’t true at runtime.
- Update symbol references to their new position

Object File

- An Object File, contains an Object Module which is just a sequence of bytes
- File Types
  - Relocatable (eg libAdd1.o)
    - Binary code/data that can be combined with other relocatable files to create an executable
    - Generated by compilers/assemblers
  - Executable (eg someExec.out)
    - Binary code/date that can be copied into memory and executed
    - Generated by linkers
  - Shared (eg libAdd1.so)
    - Special version of relocatable object file that can be loaded into memory and linked dynamically at load time or run time.
    - Generated by compilers/assemblers

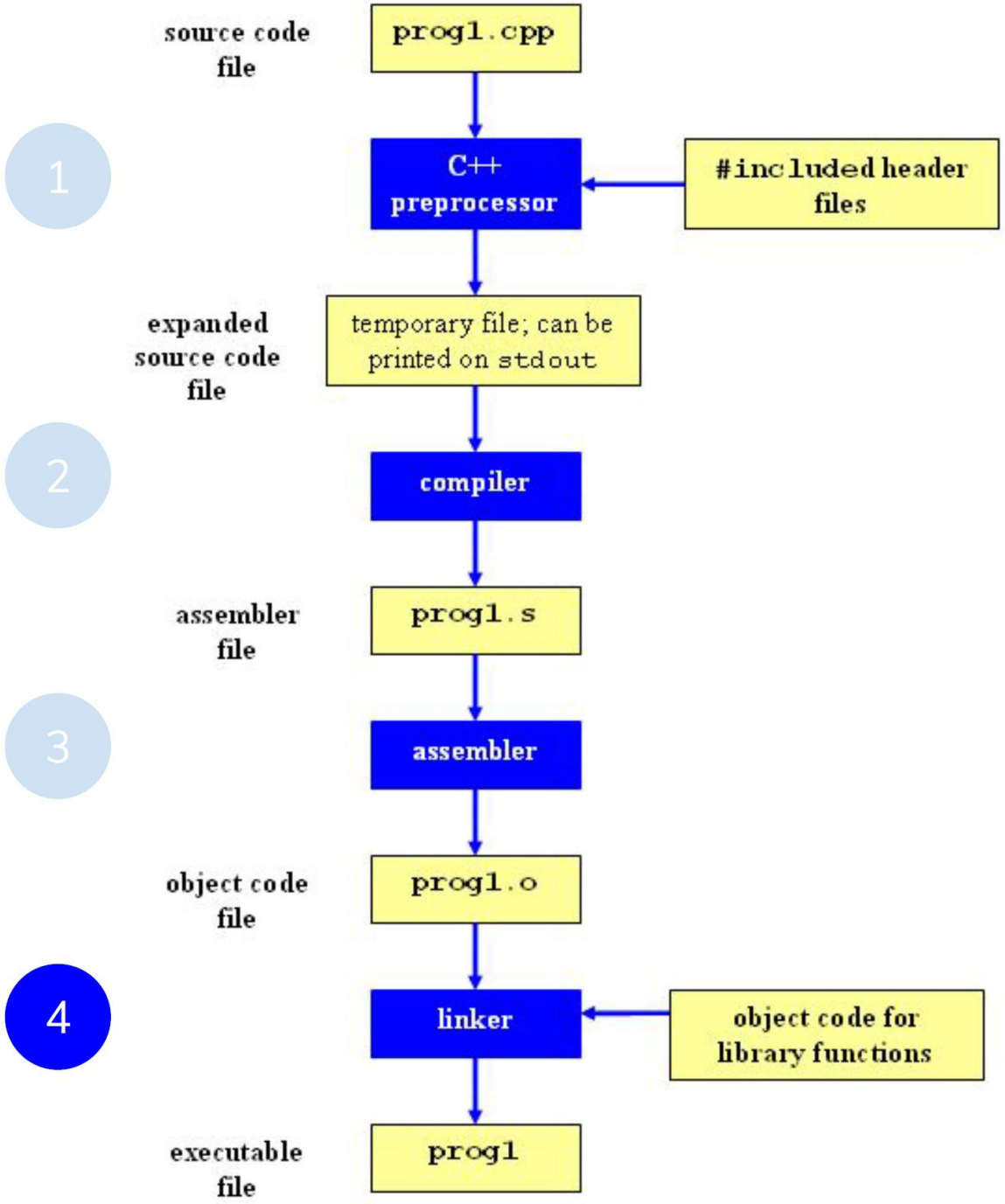
ELF

Executable and Linkable Format (ELF) is a common standard file format for object files.



Overall Compilation

- Compilers and Assemblers generate **relocatables**
- Linkers take **relocatables** and generate **executables**



Static Linking

Static Linking

- A static library is a collection of object files
  - Usually denoted by .a extension
- Static linking finds references to static libraries, and copies the relevant modules to the executable
- Static linking is done by the linker as the last step in compilation.
  - NOTE – this means static linking is only done once at compile time!

Using static libraries

- In Unix, static libraries are stored on disk in archive format (*.a).
  - Use the `ar` command to convert objects to archive files
  - Examples – libc.a (C standard library) or libm.a (C math library)
- To compile statically over shared, specify using –static
  - GCC automatically searches for predefined static libraries
  - Or, manually specify with the –L and –l parameters
  - `gcc –static main.c –L`*`/path/to/libraryDir`* `–l`*`library`*

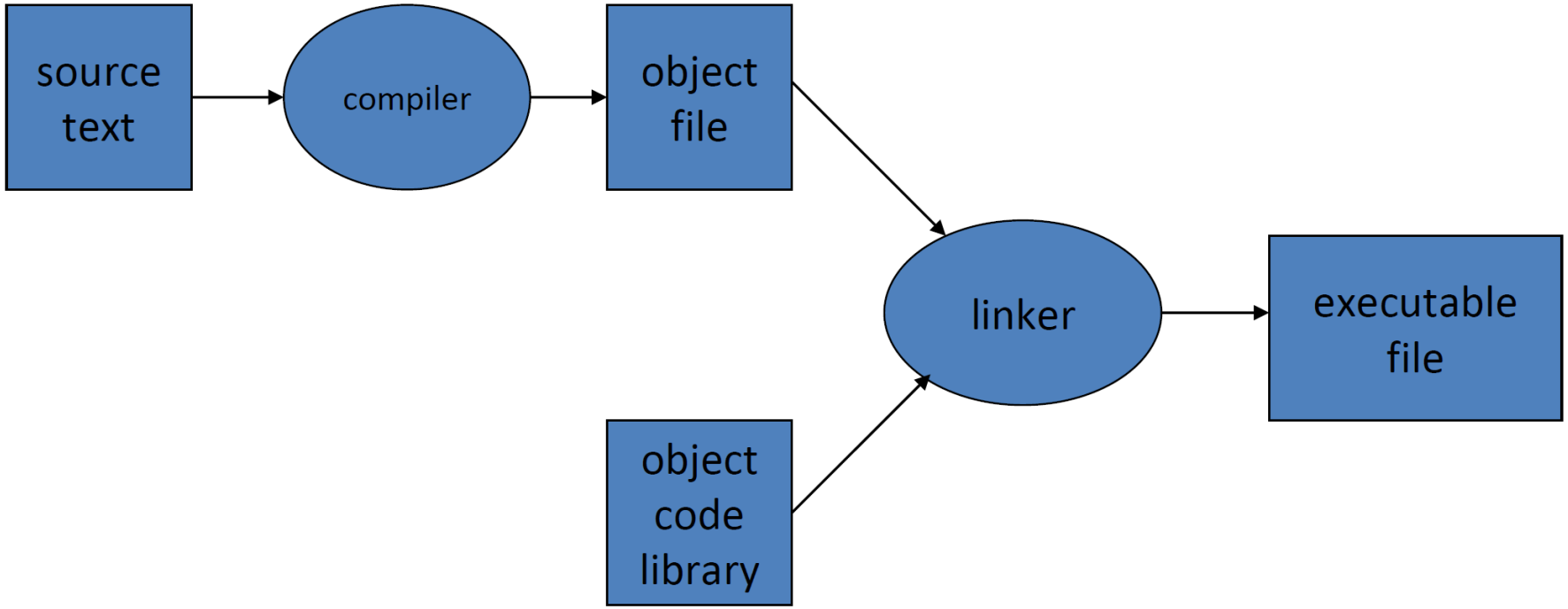
Example

```
#Create object file of library
gcc -c libAdd1.c -o libAdd1-static.o

#Create static archive file
ar rcs libAdd1-static.a libAdd1-static.o

#Compile and statically link file
gcc -c useAdd1.c -o useAdd1.o
gcc -o add1_static useAdd1.o libAdd1-static.a
```

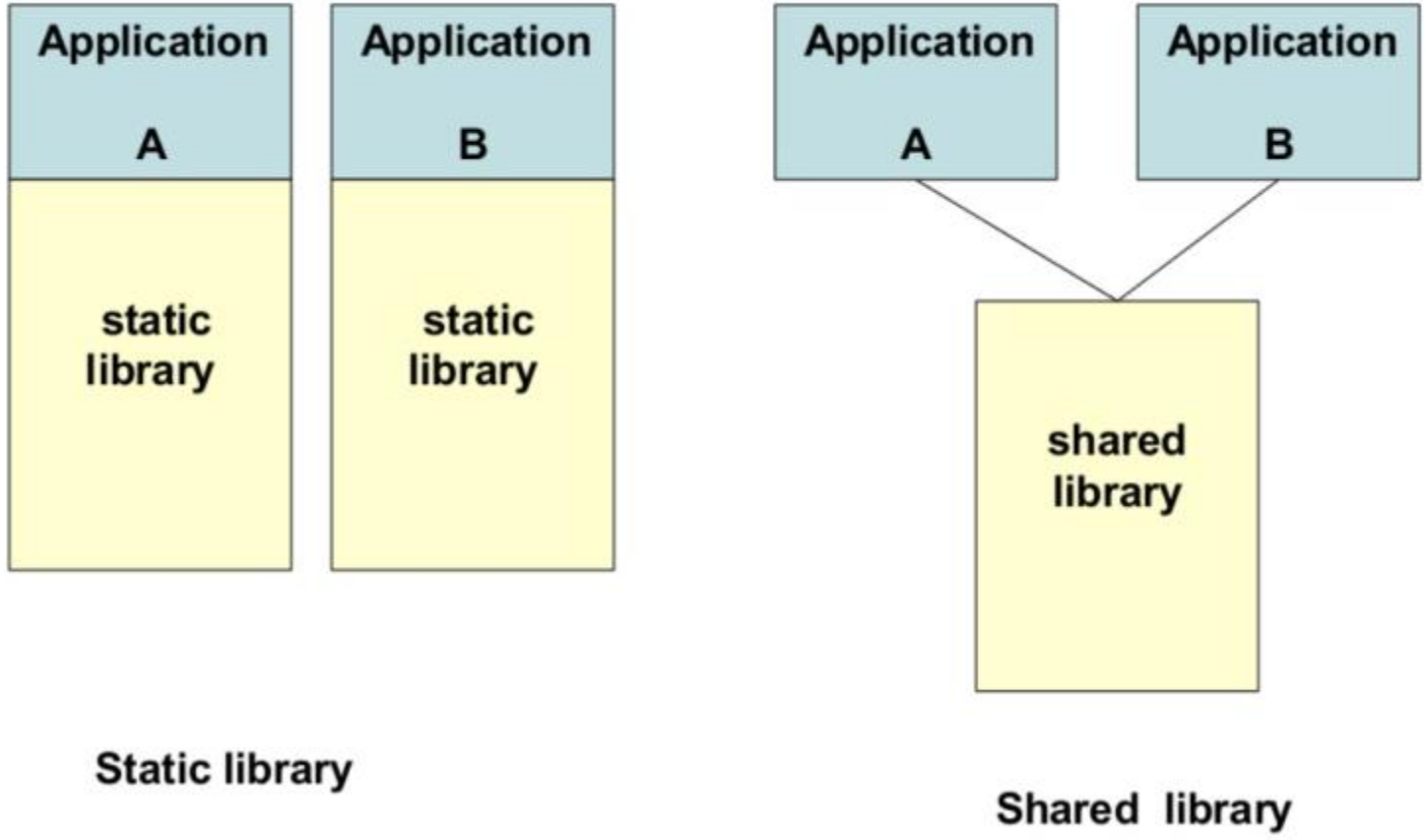
Static Linked Executables



A previously compiled collection of standard program functions

Dynamic Linking

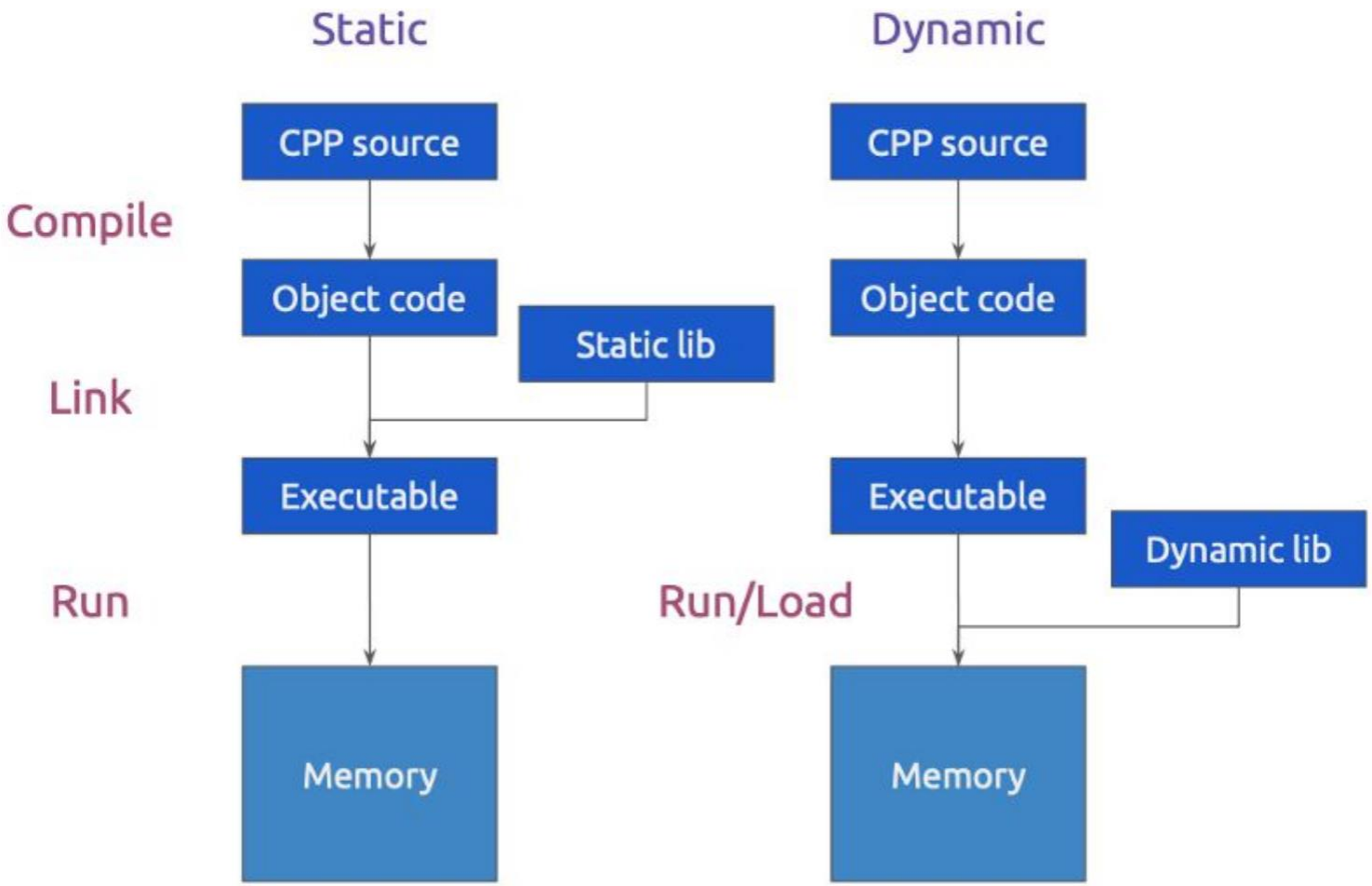
Static vs Dynamic Linking at a Glance



Shared Libraries and Dynamic Linking

- At compile time, we just copy references to the library instead of the actual library code itself
- At **runtime**, a shared library is loaded to an arbirtrary memory location and linked with the running program.
  - This means the linking step takes place at **runtime**
- Several executables will all share/use the exact same library copy
  - Big difference from static linking
- This whole process above is Dynamic Linking

Static vs Dynamic Linked Executables



Shared Library Naming

- If there are multiple versions of a library on the system, how do we know which version to use?
- Naming is divided into 3 types:
  - Real name – *libLibrary.so.major.minor* (e.g. libadd1.so.2.1.3)
    - The actual file
  - Soname – *libLibrary.so.major* (eg. libadd1.so.2)
    - Symbolic link to major version
  - Linker name – *libLibrary.so* (eg. Libadd1.so)
    - Symbolic link to latest soname

Major vs Minor Release

- Major (eg. libadd1.so.2 vs libadd1.so.1)
  - The new version's API is not backwards compatible
  - Incorporating the new version will break your current code
- Minor (eg. libadd1.so.1.1 vs libadd1.so.1.0)
  - Still backwards compatible
  - May still change/add functionality but will not break current code

Dynamic Compiling Example

```
#Compile with fpic to output "position independent code"
gcc -fPIC -c add1.c

#Compile as shared library, and pass name to linker
#-Wl,-soname paramater says what filename the executable should link on
#   We choose the major version symbolic link, libAdd1.so.1
gcc -shared -Wl,-soname,libAdd1.so.1 -o libAdd1.so.1.0 add1.o

#Create symbolic links to original shared object file
ln -sf libAdd1.so.1.0 libAdd1.so.1
ln -sf libAdd1.so.1 libAdd1.so

#Compile executable
# -L flag to give Library Directory (where library stored)
# -l for library name. Assumes name structure is lib[name].so or .a,
#    so you only provide [name] to -l flag
gcc -c useAdd1.c -o useAdd1.o
gcc useAdd1.o -L${PWD} -lAdd1 -o add1_dynamic
```

Both Linker and Loader need to find library

- LD_LIBRARY_PATH variable tells the loader where to look for libraries
  - The path to the library we just created, is NOT in there.

- How to fix? Two options
  1. Modify the LD_LIBRARY_PATH Variable permanently, or every time we run the program(not ideal)

```
LD_LIBRARY_PATH=.
export LD_LIBRARY_PATH
./add1_dynamic
```

  2. Tell the executable itself (at compile time) where to run shared libraries from. Use Wl to send the ‘run path’ parameter to the linker

```
-Wl,-rpath,.
```

Dynamic Compiling Updated - rpath

```
#Compile with fpic to output "position independent code"
gcc -fPIC -c add1.c

#Compile as shared library, and pass name to linker
#-Wl,-soname paramater says what filename the executable should link on
#   We choose the major version symbolic link, libAdd1.so.1
gcc -shared -Wl,-soname,libAdd1.so.1 -o libAdd1.so.1.0 add1.o

#Create symbolic links to original shared object file
ln -sf libAdd1.so.1.0 libAdd1.so.1
ln -sf libAdd1.so.1 libAdd1.so

#Compile executable
# -L flag to give Library Directory (where library stored)
# -l for library name. Assumes name structure is lib[name].so or .a,
#    so you only provide [name] to -l flag
# -Wl,-rpath,. Tells the executable to load the library from the current directory '.' at runtime
gcc -c useAdd1.c -o useAdd1.o
gcc useAdd1.o -L${PWD} -Wl,-rpath,. -lAdd1 -o add1_dynamic
```

Pros and Cons of Dynamic Linking

- Pros
  - The executable itself is smaller in size
  - The memory footprint of the library itself is amortized across all programs using the same library
  - When the library is changed, you only need to recompile the single library. You don’t need to recompile all of the applications that use the library
- Cons
  - Programs usually startup and run slower, because linking is done at the fly so there are extra steps involved at load and runtime.
  - The library can be missing, or it can be the wrong version

NOTE – Static linking is mainly the opposite of these (has larger memory footprint, needs to be recompiled for any change, but - runs faster and once compiled is mostly guaranteed to run).

Example of Updating Library Change add1() to add 5

1. Modify the source code add1.c
2. Recompile the library into a new version
3. Update the symbolic links
4. Original application will still work with new library code!

```
#Compile with fpic to output "position independent code"
gcc -fPIC -c add1.c

#Compile as shared library, and pass name to linker
#Increment minor version from 1.0 to 1.1
gcc -shared -Wl,-soname,libAdd1.so.1 -o libAdd1.so.1.1 add1.o

#Update symbolic links
ln -sf libAdd1.so.1.1 libAdd1.so.1
ln -sf libAdd1.so.1 libAdd1.so
```

Library Hierarchy in Linux