CSC 230 - Summer 2018

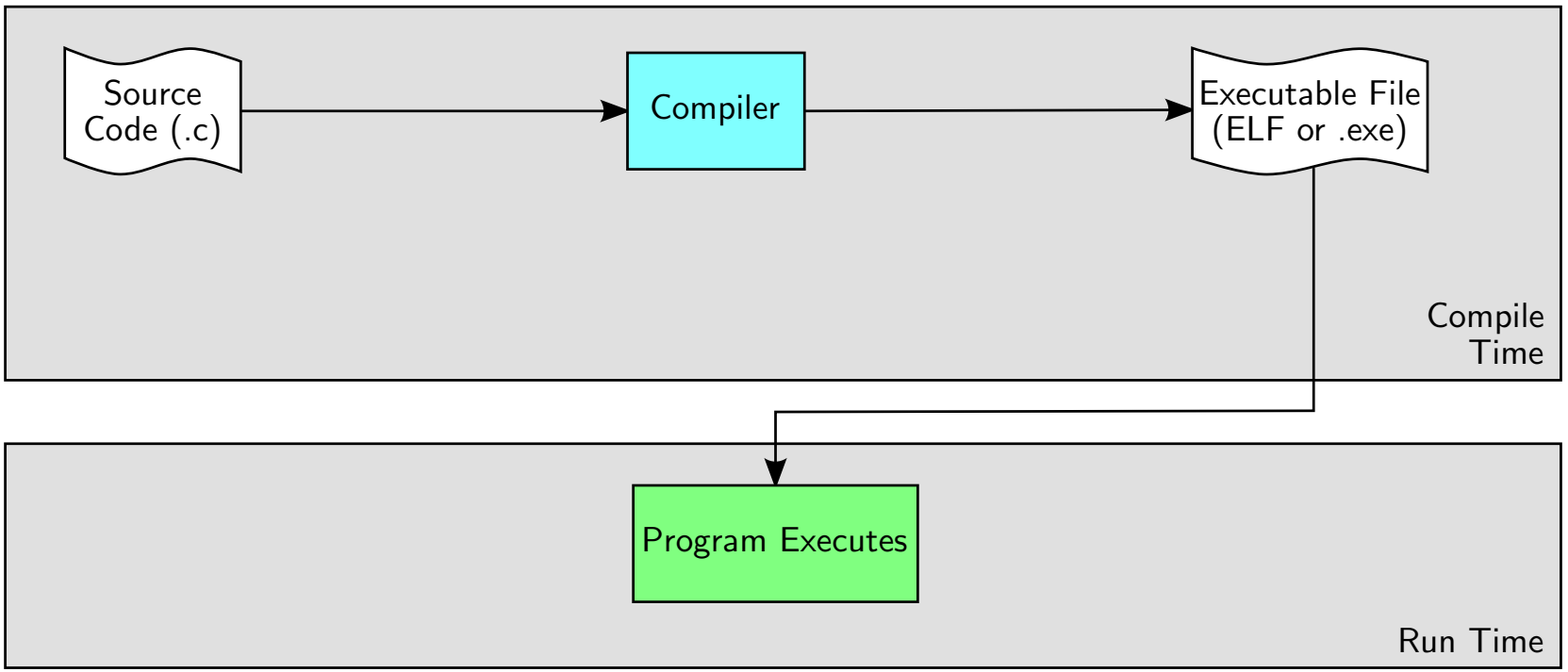
Compiling, Linking and Loading

Bill Bird

Department of Computer Science
University of Victoria

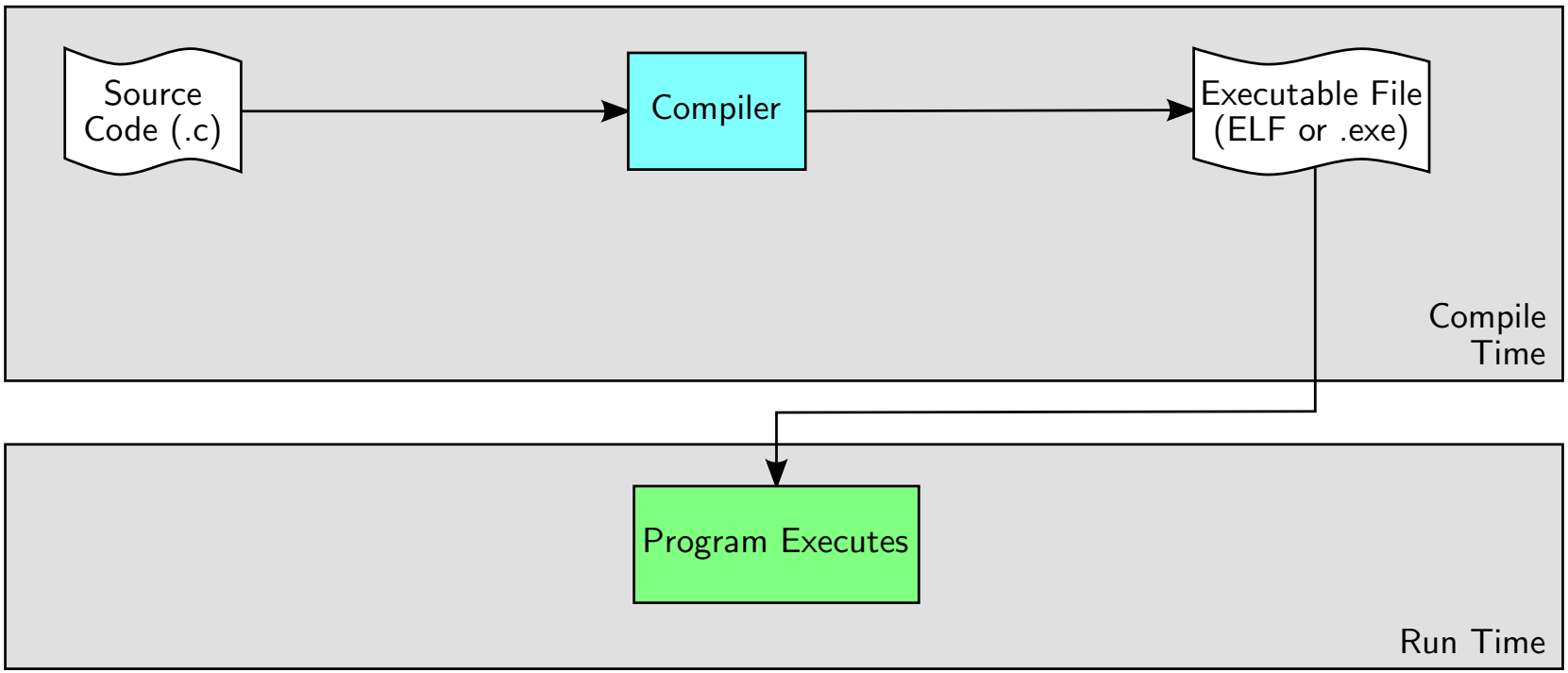
July 30, 2018

The Build Process (1)



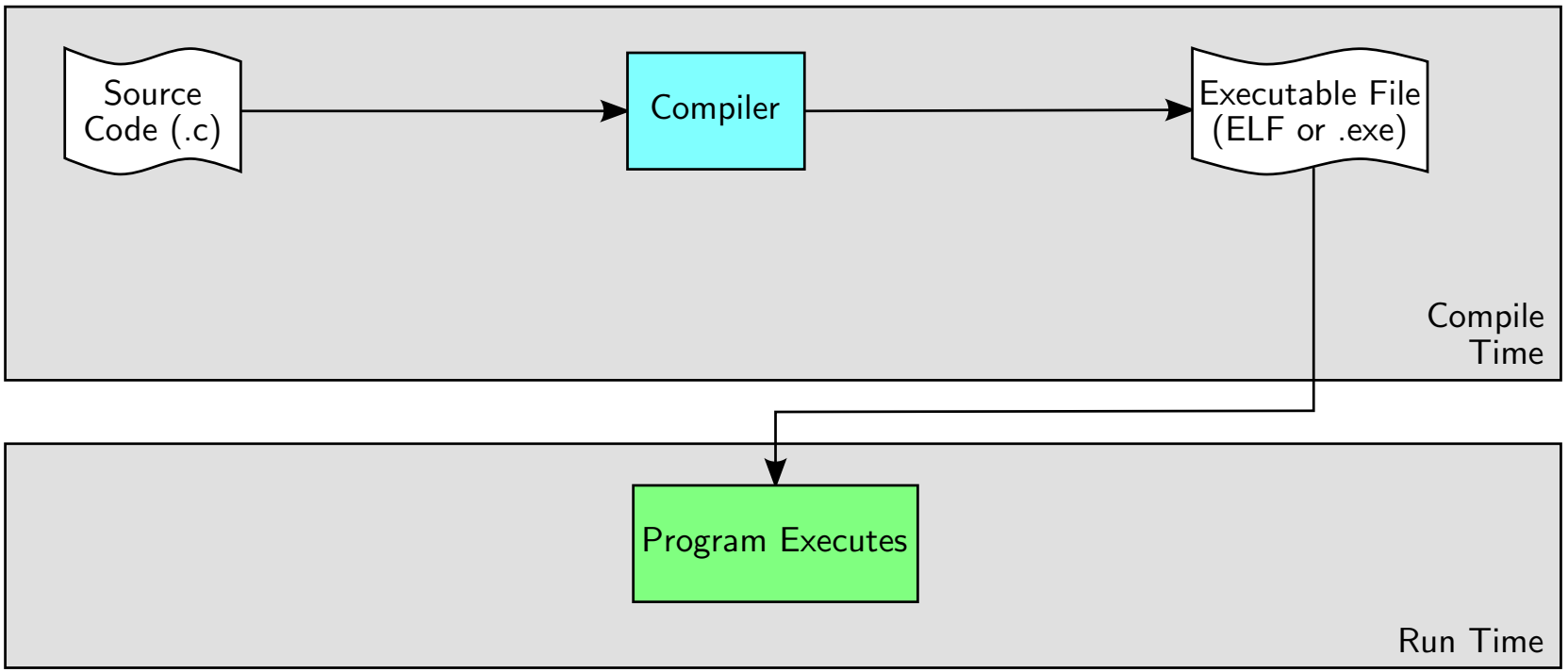
Compiled languages like C and C++ are converted from source code into binary machine instructions before run time. When the program is run, the executable file is copied into memory and executed.

The Build Process (2)



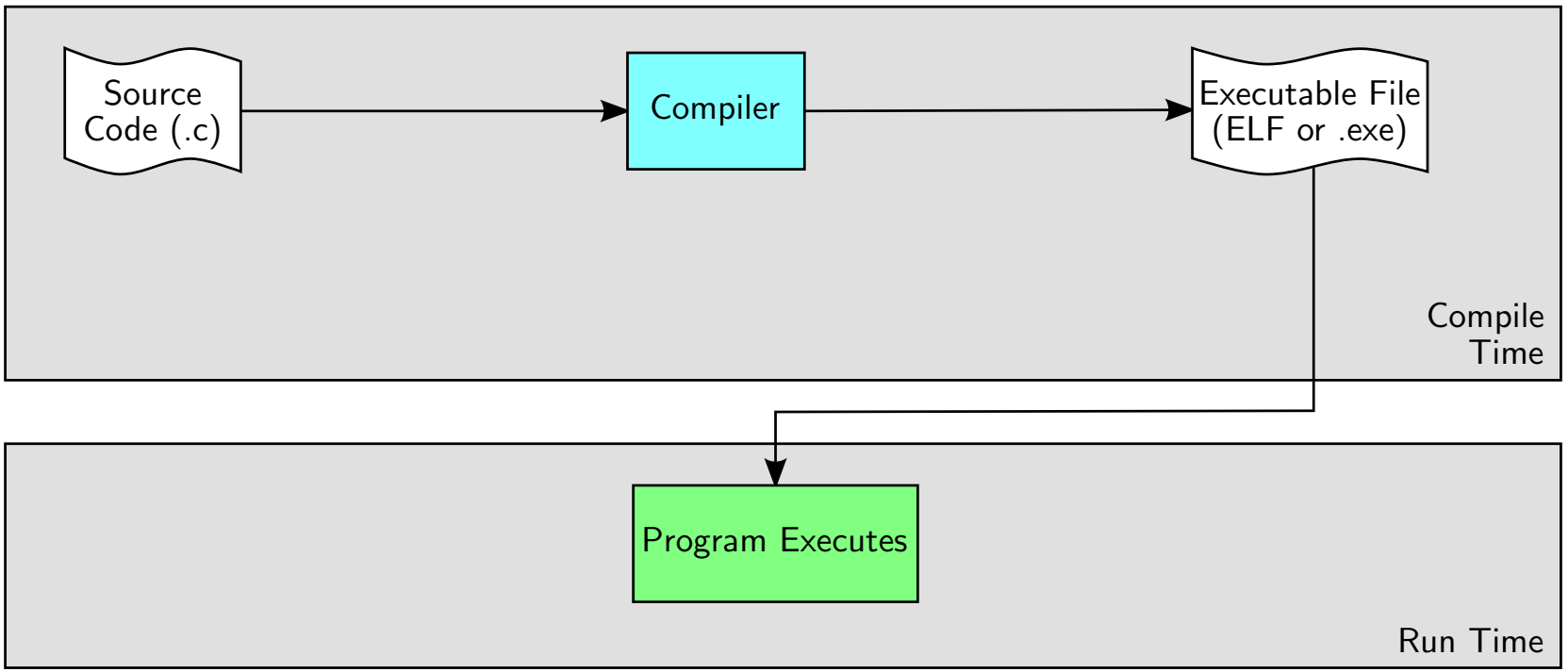
Interpreted languages (like Python and Javascript) and virtual machine based languages (like Java) follow different build processes that are not part of this course.

The Build Process (3)



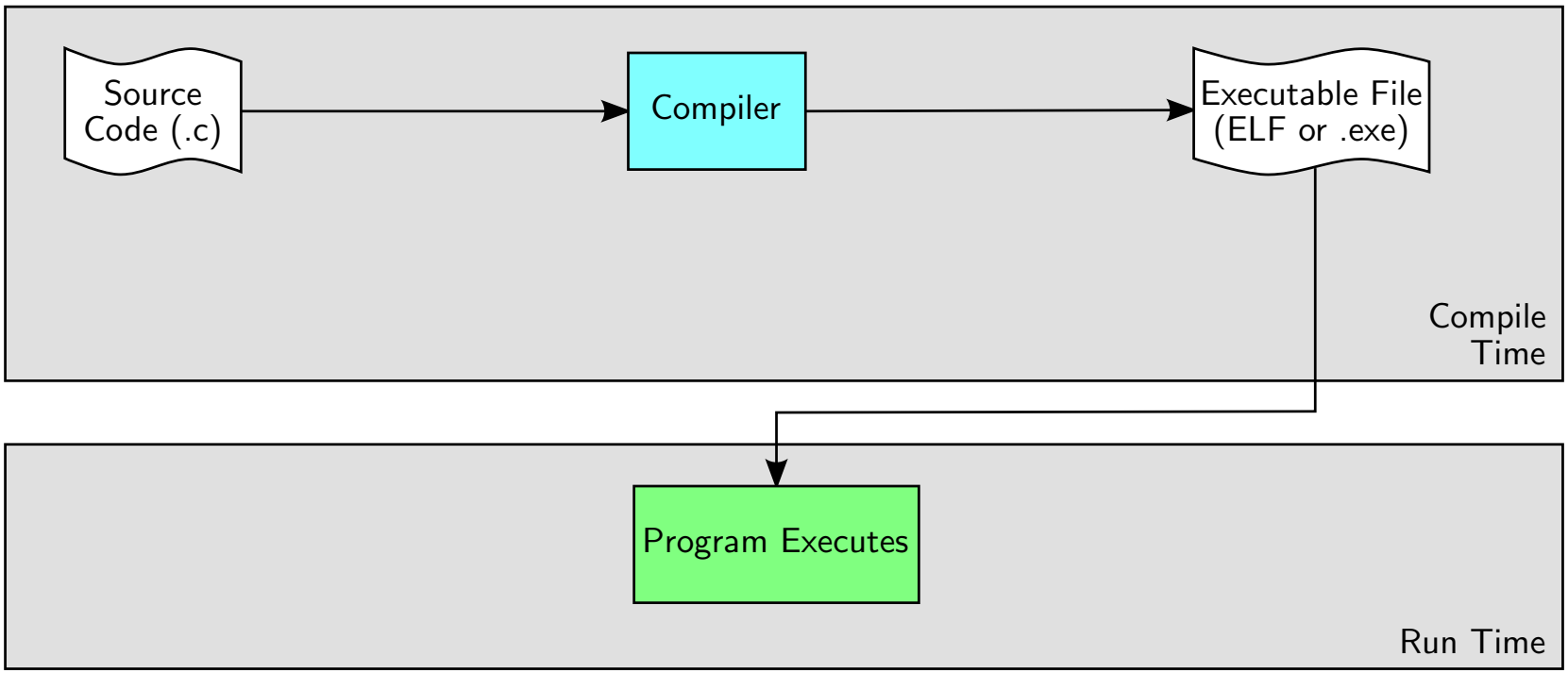
Simple C programs can be completely built using a single command (such as one invocation of gcc). As a result, the build process appears to reflect the simplified model shown above.

The Build Process (4)



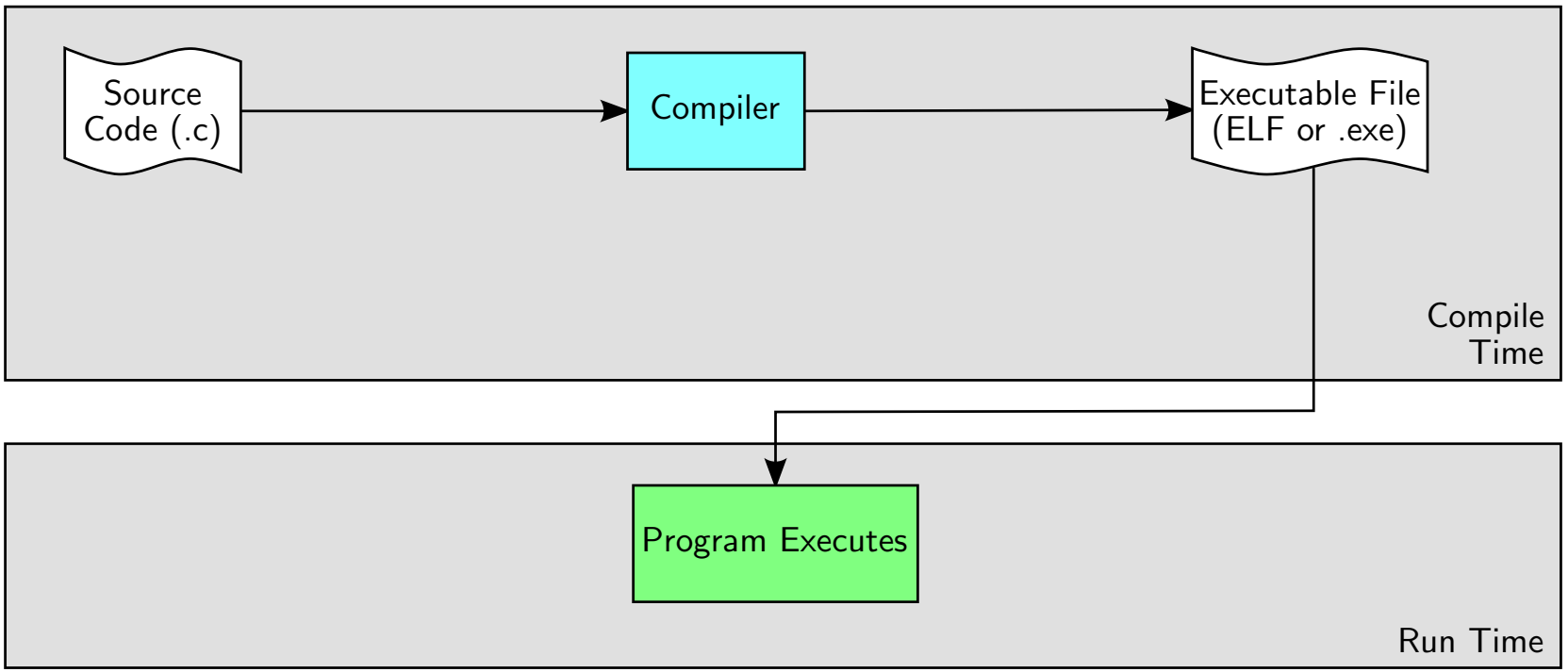
A command like '`gcc -o my_executable my_source_file.c`' on a modern Unix-based system will compile `my_source_file.c` into an executable called `my_executable`.

The Build Process (5)



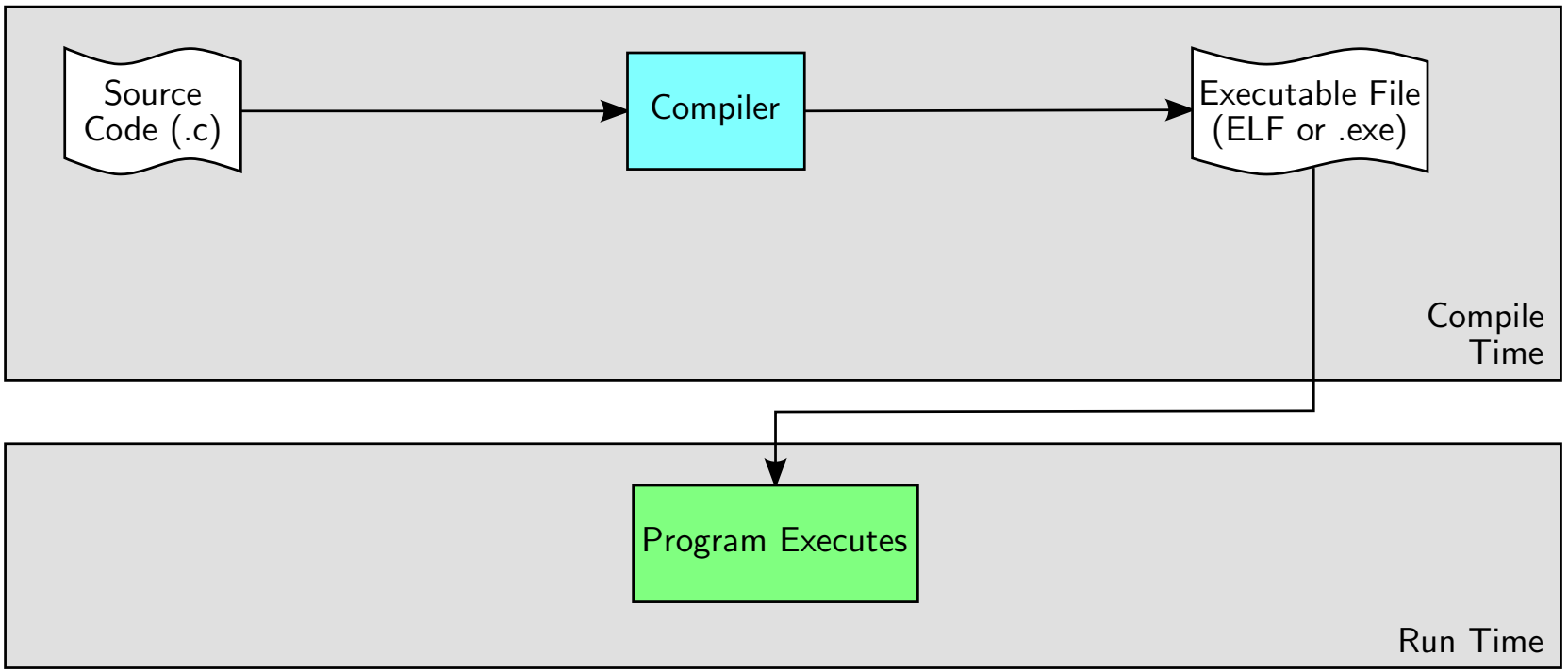
Executable files are stored in a format called ELF (Executable and Linkable Format) on most Unix-like systems. Windows executables (with the `.exe` extension) use a standard called PE (Portable Executable).

The Build Process (6)



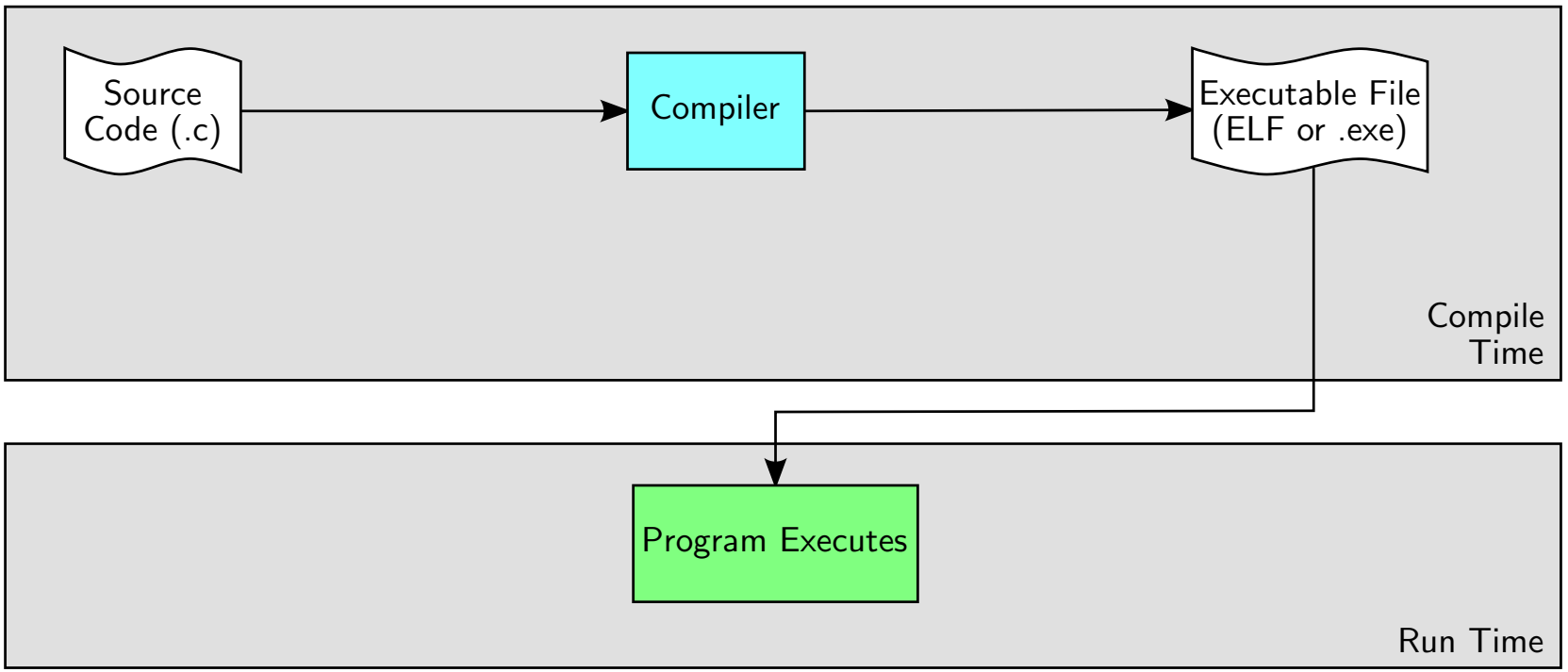
When the program is run, the executable file is loaded into memory by the OS. The OS then jumps into hidden initialization code added to the program by the compiler, which calls `main()`.

The Build Process (7)



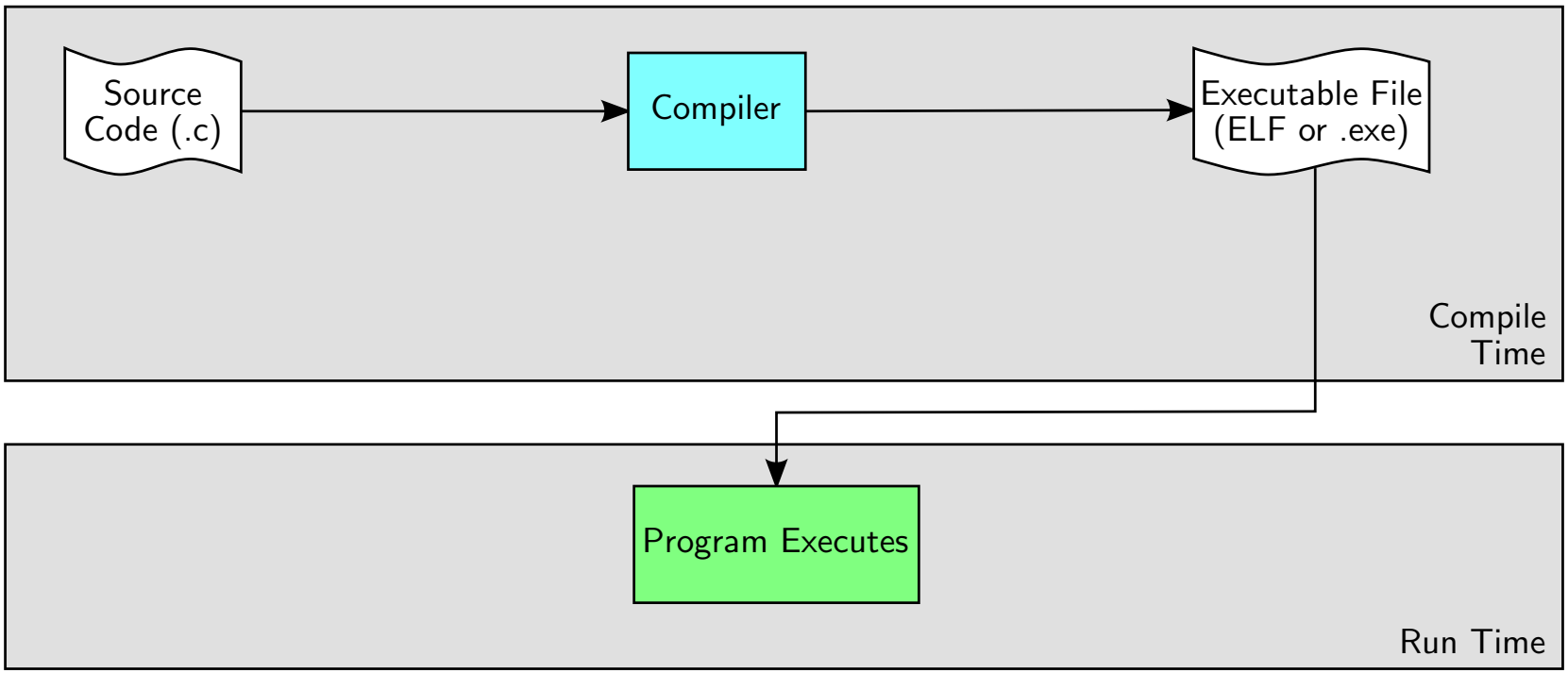
Question: When the program is loaded into memory, how are functions like `printf` (or other library functions) provided?

The Build Process (8)



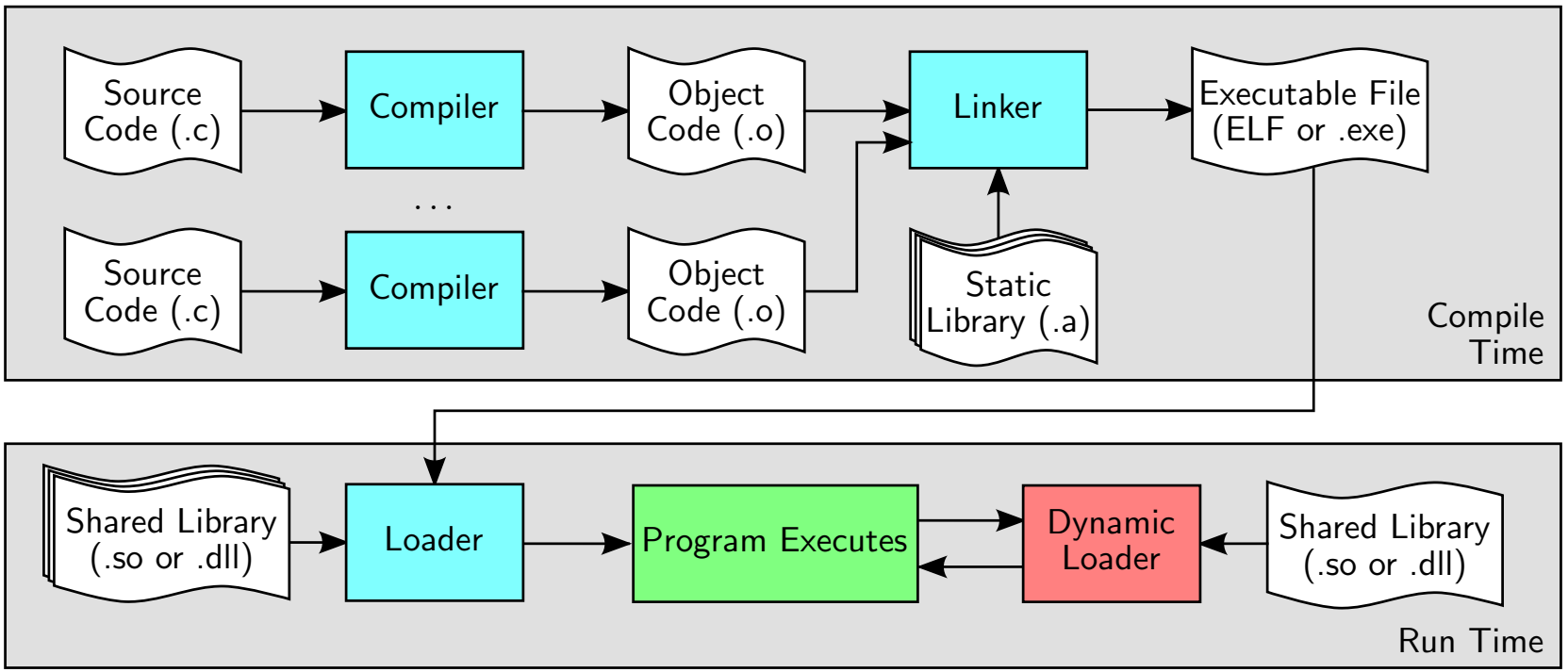
All function calls in C code (including calls to standard library functions) are treated equally by the compiler (and translated into appropriate assembly instructions, like the AVR `CALL` instruction). Therefore, the code for `printf` must be somehow included into the executable image of the program in memory.

The Build Process (9)



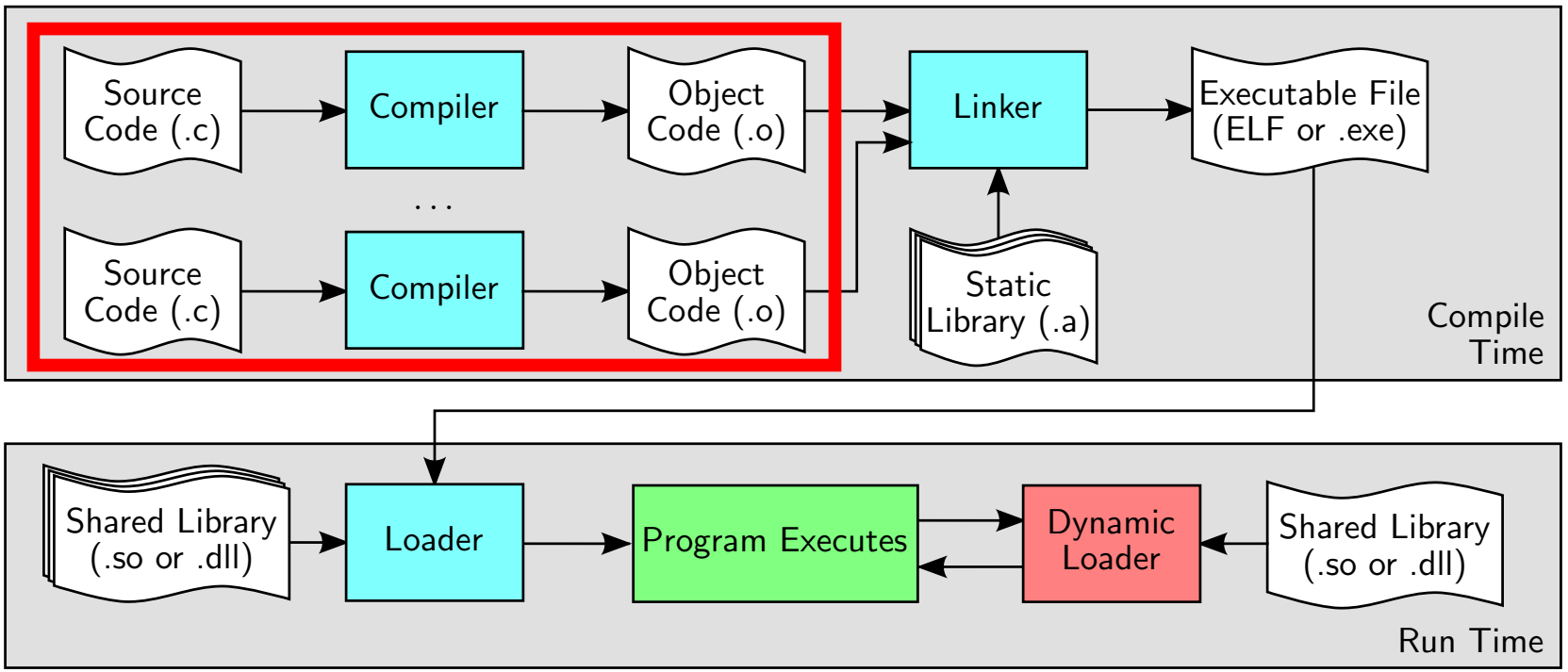
The model above is too simple to capture the actual steps followed between the initial compile command and the eventual call to the `main` function at runtime.

The Build Process (10)



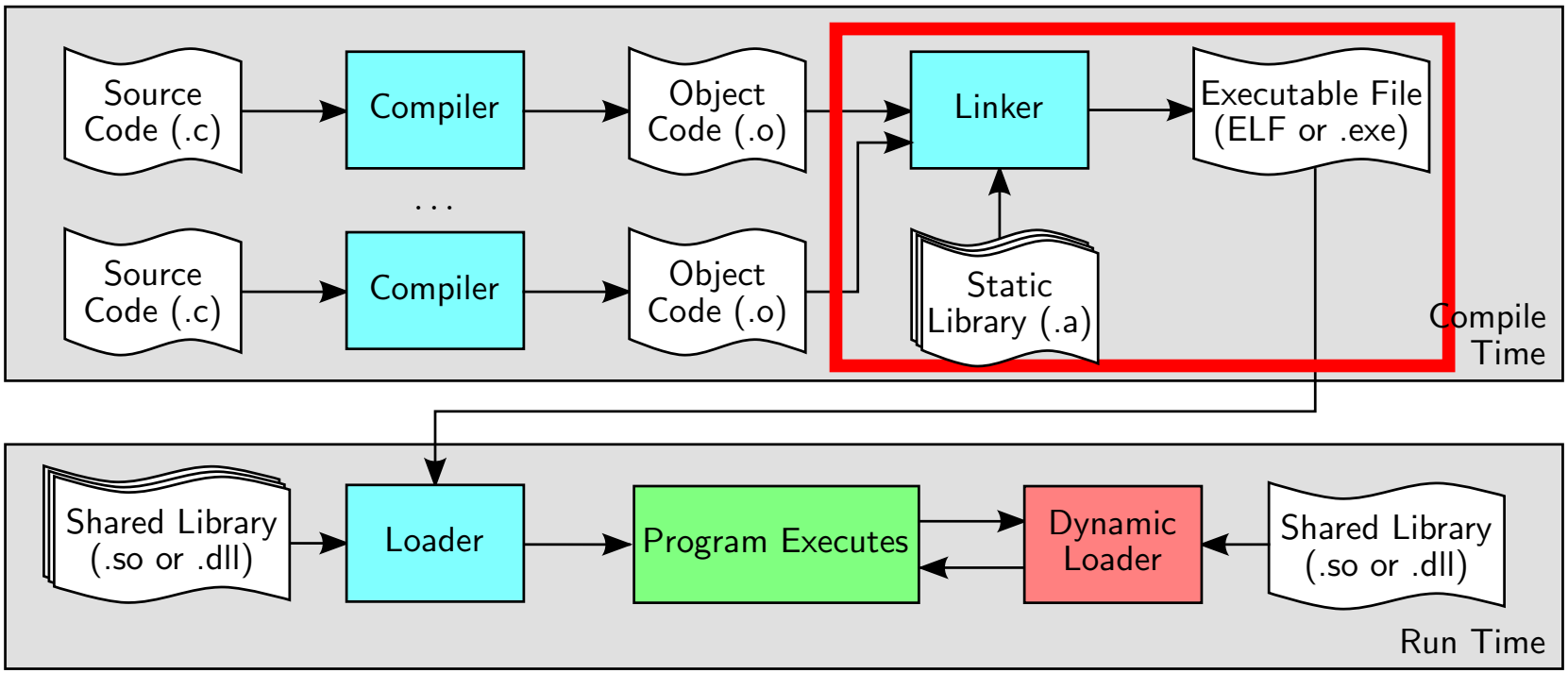
The actual build process for a compiled language is more complicated.

The Build Process (11)



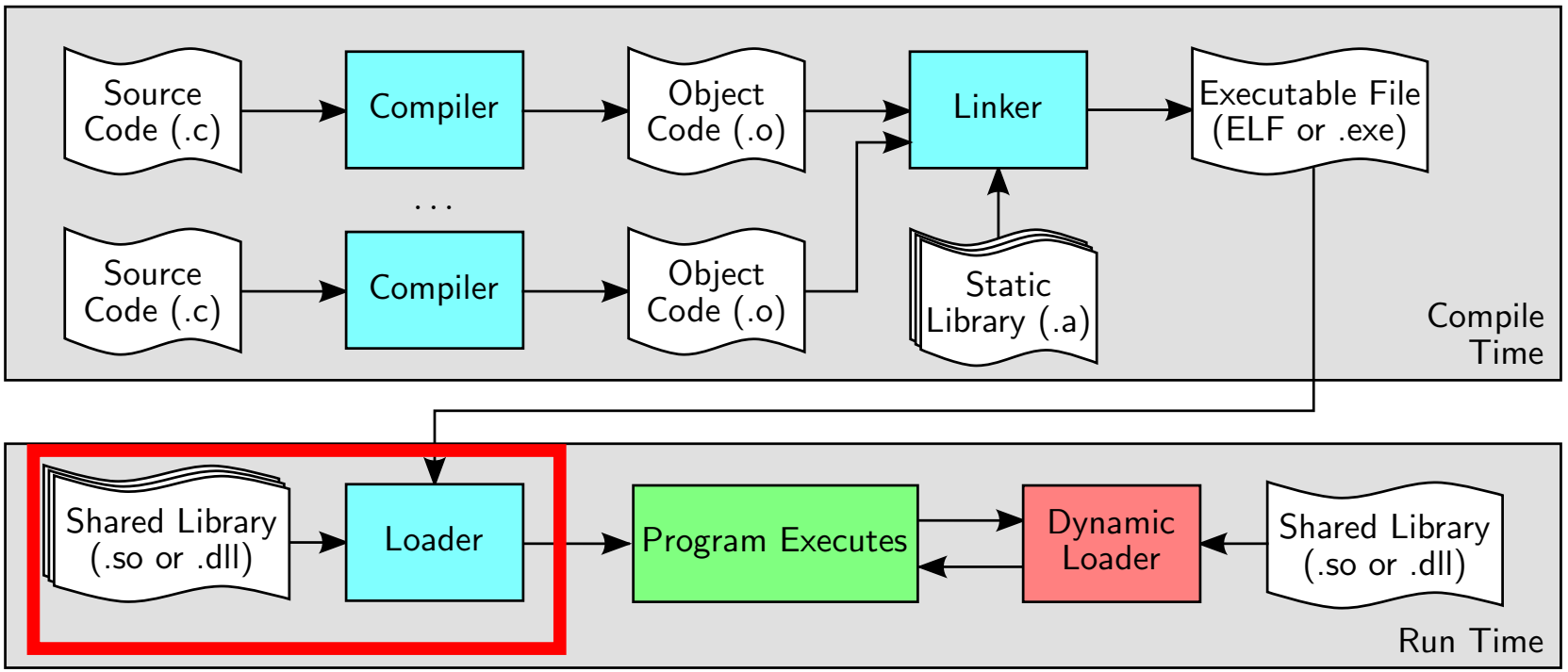
In the full build process, the **compiler** only manages the process of converting each individual code unit (usually one `.c` file) to binary machine instructions. In the C compilation process, the compiled form of each code unit is called **object code** and has the `.o` extension.

The Build Process (12)



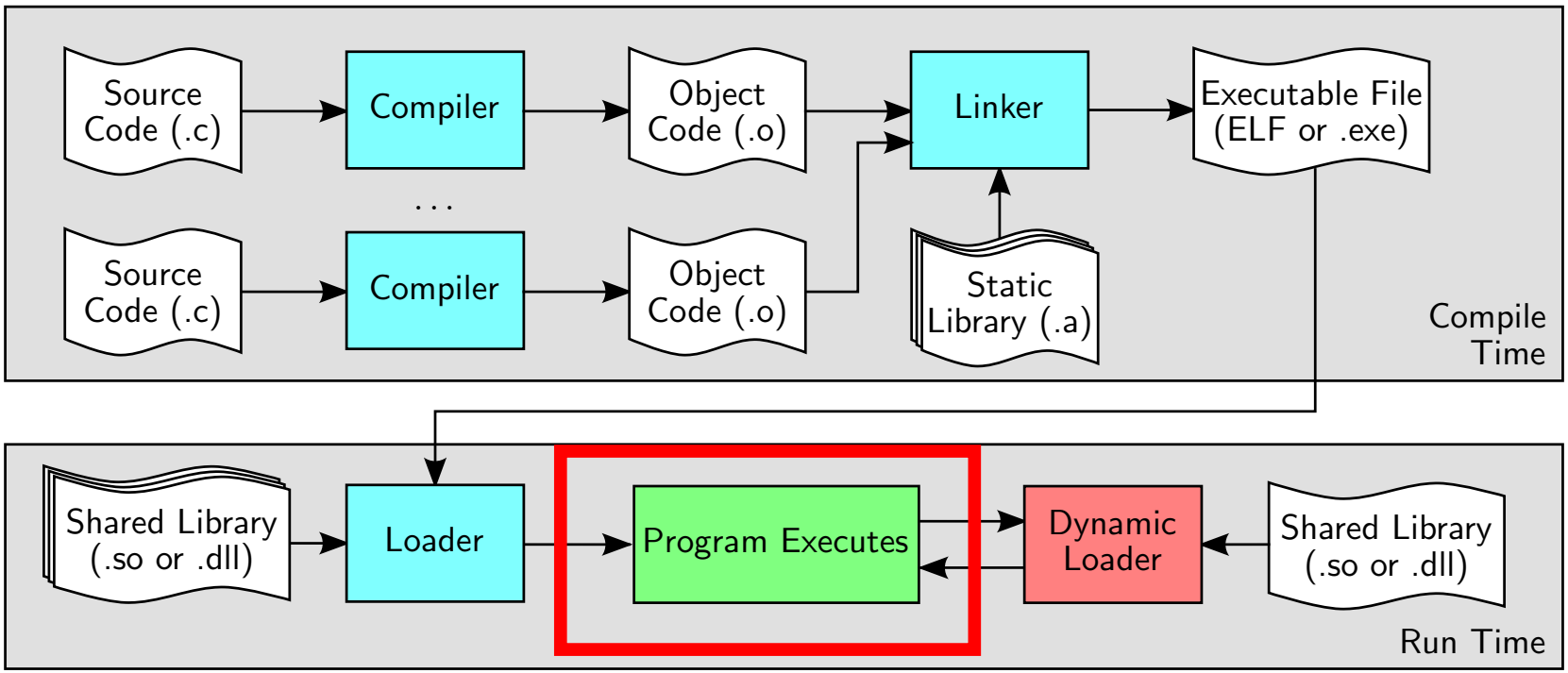
A separate process, called the **linker**, is used to join the various object files into a unified executable package. External code and data, in the form of a **static library**, may also be included into the executable.

The Build Process (13)



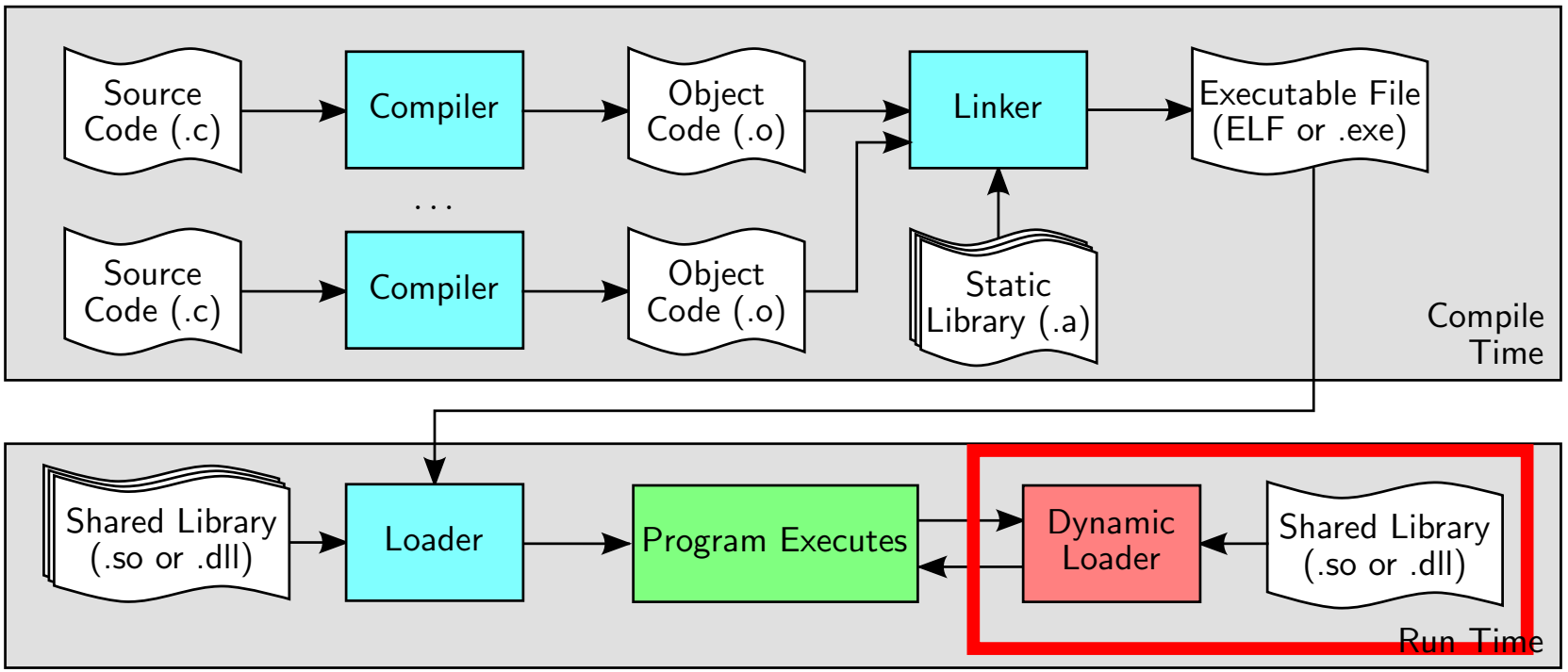
The finished executable file may still be 'incomplete' in some ways. At run time, the OS uses a **loader** to prepare the executable to run. The loader behaves similar to the linker, but at run time, and may load one or more **shared libraries** to provide code or data to the program.

The Build Process (14)



After all libraries have been prepared, the program starts running. When `main` finally begins, all of the necessary functions (e.g. `printf`) have been found and loaded into memory somewhere, so all `CALL` instructions will have a definite destination.

The Build Process (15)



However, the program may also **dynamically load** additional libraries as needed. This is done at the program's discretion (and the loading process is written by the programmer, not automatically generated by the compilation process as in the initial load stage).

Multi File Programs (1)

```c
//Function declaration
int F(int x, float y);

int G(int z){
        return z*2;
}

int main(){
        int A = F(6,10);
        int B = G(A);
        return 0;
}
```

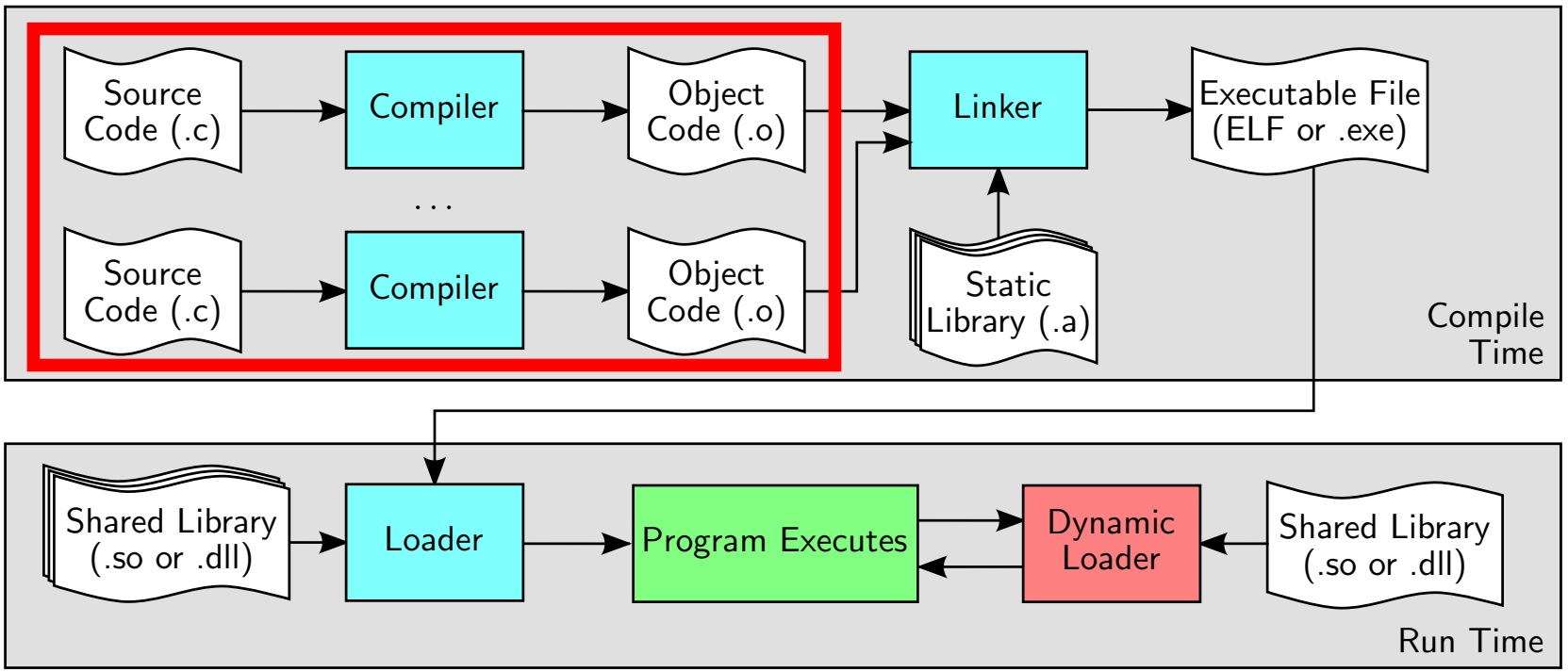
main.c

```c
int F(int x, float y){
        int result = x*x - y;
        return result;
}
```

a_function.c

Consider the two C source files above. Note that the program in `main.c` isn't actually useful (it generates no real output).

Multi File Programs (2)



Each of the `.c` files is compiled separately into binary machine code. Internally, the compiler may use text assembly code as an intermediate representation, but is not required to do so.

Multi File Programs (3)

```
//Function declaration
int F(int x, float y);

int G(int z){
        return z*2;
}

int main(){
        int A = F(6,10);
        int B = G(A);
        return 0;
}
```

main.c

```
int F(int x, float y){
        int result = x*x - y;
        return result;
}
```

a_function.c

The compiler will therefore have no knowledge of the contents of a_function.c while compiling main.c (and vice versa). Instead, when main.c is compiled, the compiler will assume that the function F will be incorporated later.

Multi File Programs (4)

```
//Function declaration
int F(int x, float y);

int G(int z){
        return z*2;
}

int main(){
        int A = F(6,10);
        int B = G(A);
        return 0;
}
```

main.c

```
int F(int x, float y){
        int result = x*x - y;
        return result;
}
```

a_function.c

When the main.c file is converted into the binary representation in main.o, the call to F is represented by a placeholder instruction (like RCALL .+0, where the branch destination is left blank).

Multi File Programs (5)

```
//Function declaration
int F(int x, float y);

int G(int z){
        return z*2;
}

int main(){
        int A = F(6,10);
        int B = G(A);
        return 0;
}
```

main.c

```
int F(int x, float y){
        int result = x*x - y;
        return result;
}
```

a_function.c

Later, the linker will modify the `RCALL .+0` instruction to insert the correct address of `F` when the code for `F` is incorporated into the executable.

Multi File Programs (6)

```
//Function declaration
int F(int x, float y);

int G(int z){
        return z*2;
}

int main(){
        int A = F(6,10);
        int B = G(A);
        return 0;
}
```

main.c

```
int F(int x, float y){
        int result = x*x - y;
        return result;
}
```

a_function.c

The address of the placeholder `RCALL .+0` instruction is called a **relocation** and is added by the compiler to a **relocation table** in the binary file to allow the linker to identify it later and match it to the function `F`.

Multi File Programs (7)

```
//Function declaration
int F(int x, float y);

int G(int z){
        return z*2;
}

int main(){
        int A = F(6,10);
        int B = G(A);
        return 0;
}
```

main.c

```
int F(int x, float y){
        int result = x*x - y;
        return result;
}
```

a_function.c

The relocation table of each .o file will contain entries for every instruction that requires a symbol name (such as a global variable or function) that is not found in the original source file.

Multi File Programs (8)

```
//Function declaration
int F(int x, float y);

int G(int z){
        return z*2;
}

int main(){
        int A = F(6,10);
        int B = G(A);
        return 0;
}
```

main.c

```
int F(int x, float y){
        int result = x*x - y;
        return result;
}
```

a_function.c

Conversely, each `.o` file also contains a **symbol table** listing every symbol in the file, to allow the linker to determine which functions and global variables are defined in each file.

Multi File Programs (9)

```
//Function declaration
int F(int x, float y);

int G(int z){
        return z*2;
}

int main(){
        int A = F(6,10);
        int B = G(A);
        return 0;
}
```

main.c

```
int F(int x, float y){
        int result = x*x - y;
        return result;
}
```

a_function.c

(The symbol table also often contains entries for each symbol needed by a relocation)

Multi File Programs (10)

To compile a `.c` file (e.g. `my_code.c`) into text assembly code:

```
gcc -S my_code.c
```

(use `avr-gcc` to compile to AVR assembly).

To compile a `.c` file directly into a `.o` file:

```
gcc -c my_code.c
```

(use `avr-gcc` to compile to AVR assembly).

To *disassemble* a binary `.o` file to assembly code (for debugging purposes only, since the representation is formatted for human readability and cannot be recompiled):

```
objdump -D my_code.o
```

(use `avr-objdump` if the `.o` file was generated by `avr-gcc`)

Multi File Programs (11)

The `objdump` program can be used to disassemble .so files (containing object code for particular source files), as well as shared library files (usually with the `.so` extension) and complete binary executables (that is, the final output of the compile process. Besides disassembling, `objdump` can be used to extract other important aspects of binary executables.

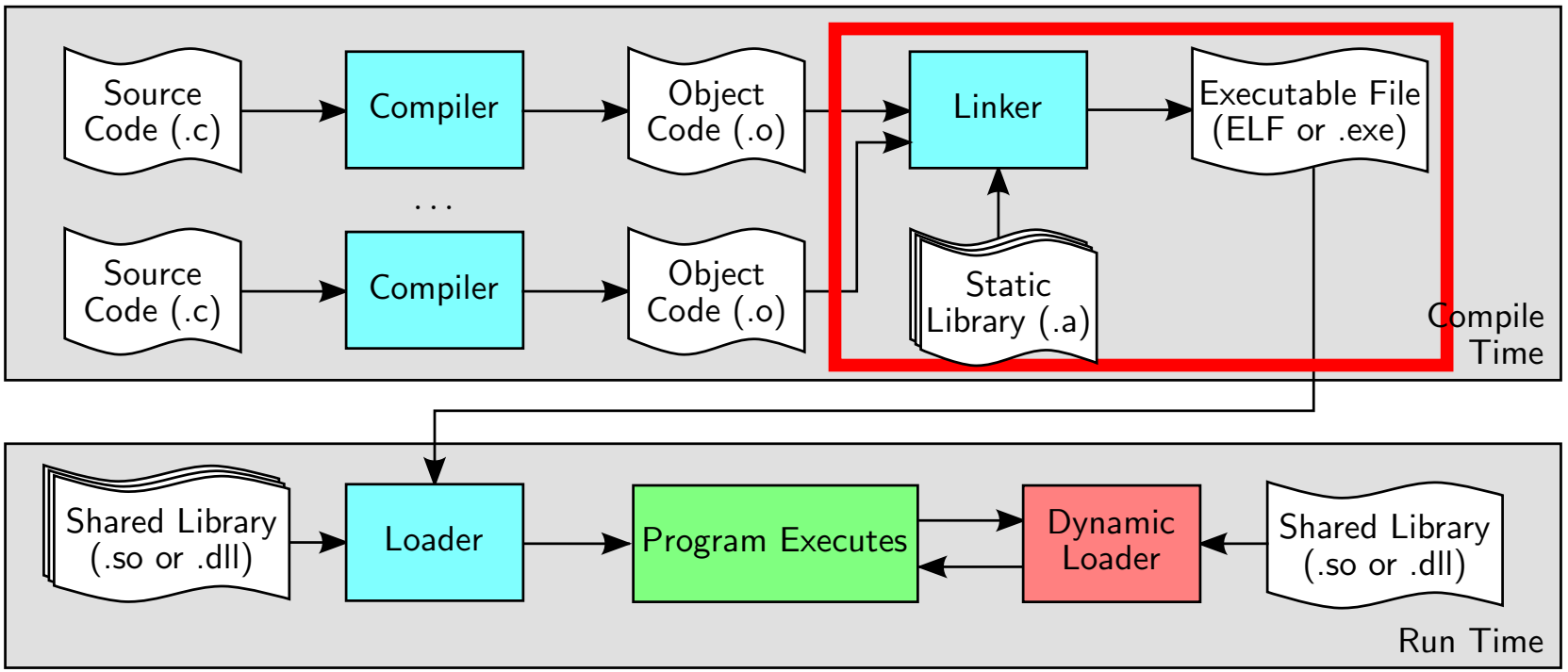
Flag	**Behavior**
`-D`	Disassemble all sections
`-r`	Extract relocation table
`-S`	Disassemble with original source code
`-t`	Extract symbol table

Multi File Programs (12)

The `-S` flag for `objdump` will print the disassembly interleaved with lines of the original source file (to make it clear which assembly instructions correspond to which lines). This is only possible if the file has been compiled with the '`-g`' flag to `gcc` (to enable debugging symbols) and if the original source file is present (so you can't use it to see the original source code of arbitrary programs.

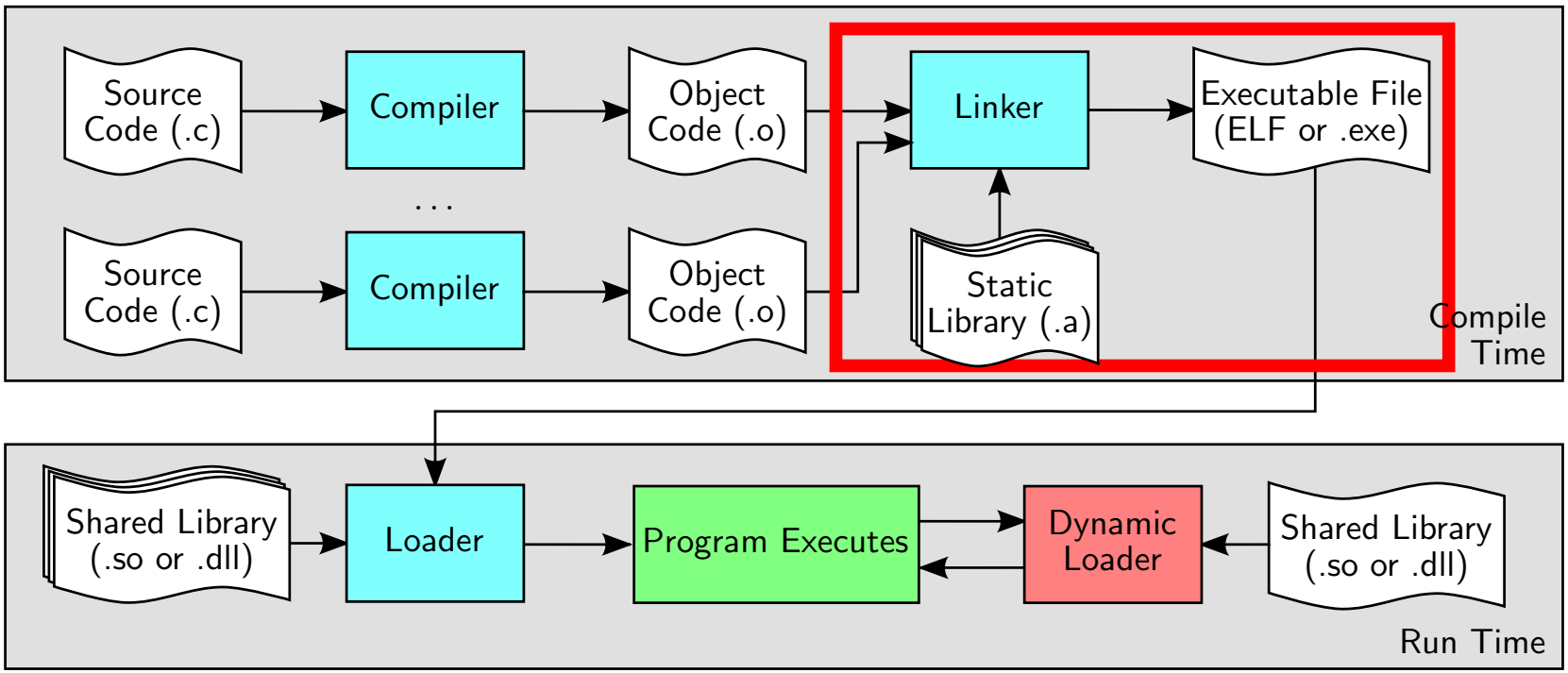
However, you can use '`-D`' to view the assembly code of any program or library to which you have read access.

Linking and Loading (1)



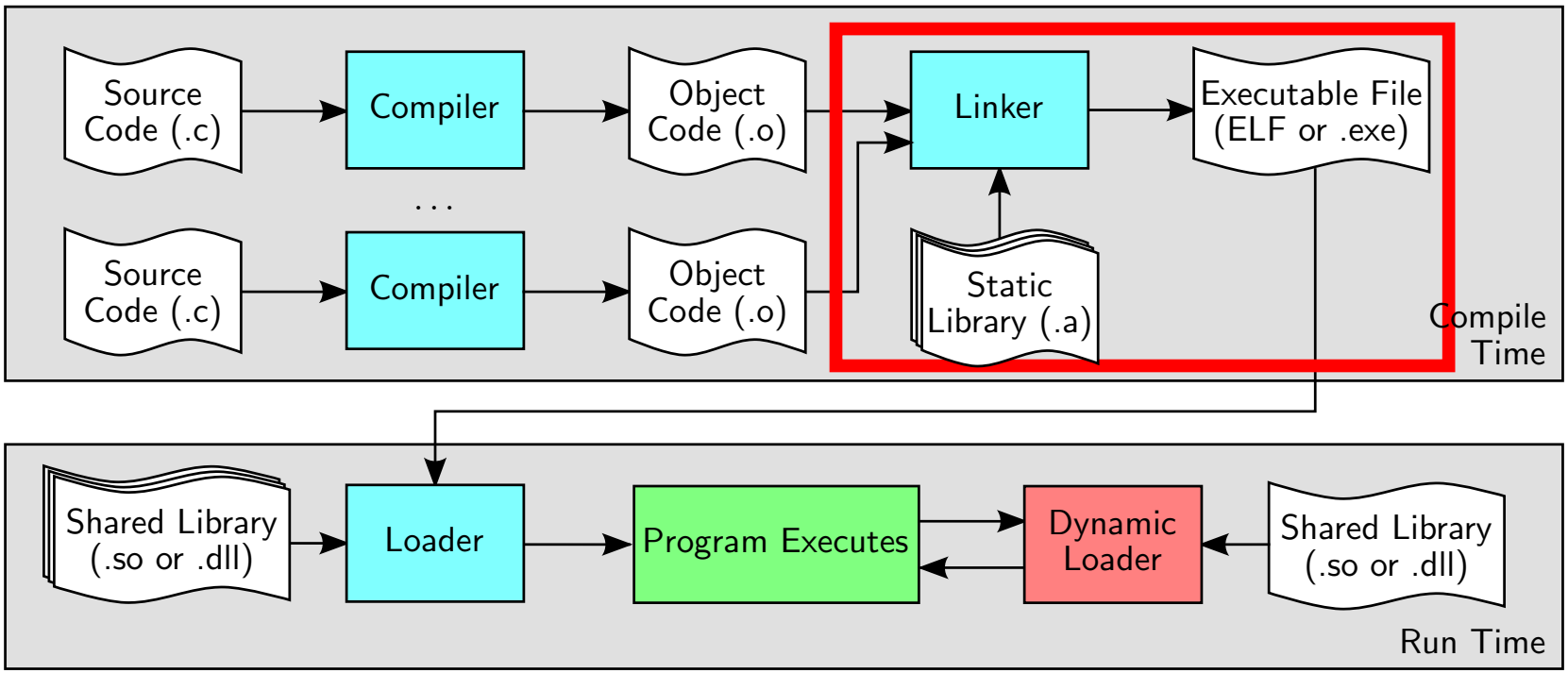
After each source file is compiled into a `.o` file, the linking process begins. The goal of the linker is to combine all of the binary code into a single file and reconcile all of the symbols and relocations among the various components.

Linking and Loading (2)



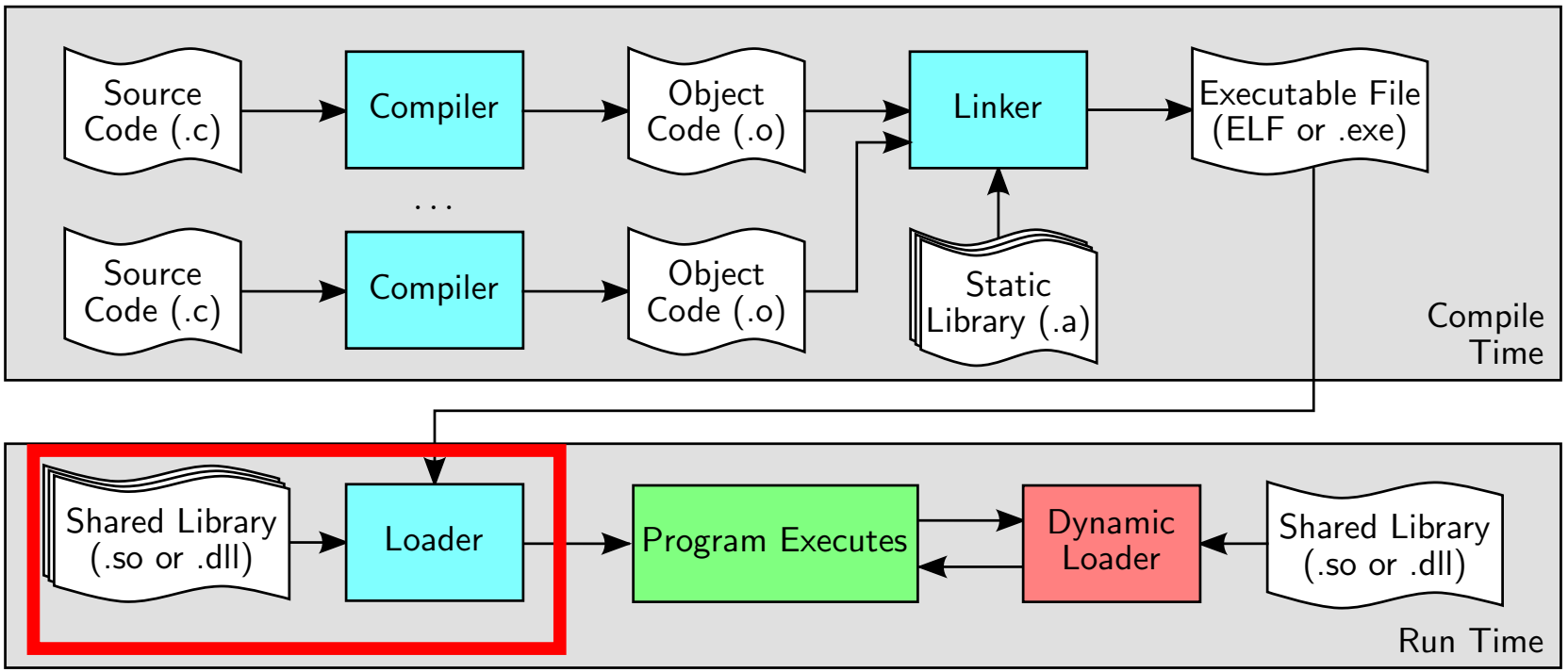
Binary code (and data) may be provided by `.o` files, but also by pre-packaged library files. Libraries which are brought in during link time are called **static libraries** and are usually just archives containing pre-compiled `.o` files.

Linking and Loading (3)



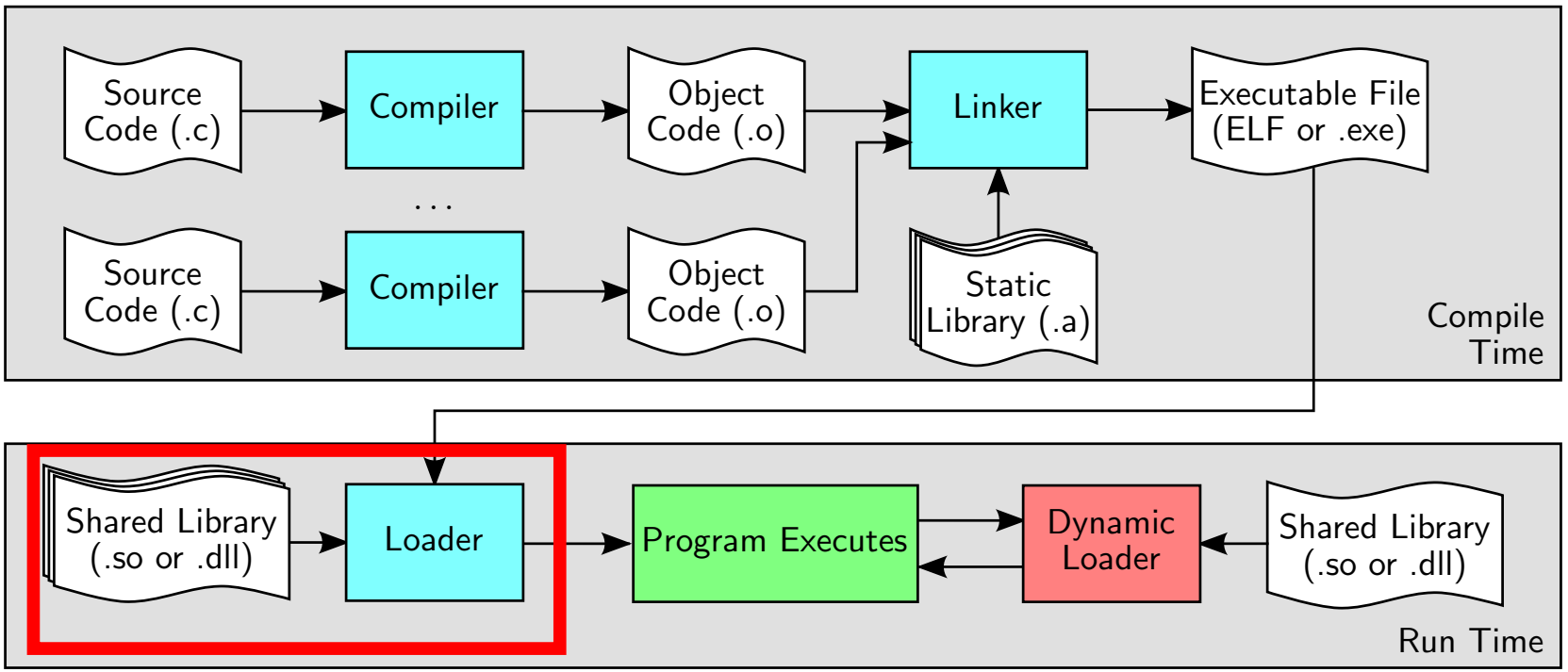
Static library code is copied directly into the executable file, which increases the size of the executable and prevents the executable from being affected by changes in the original static library.

Linking and Loading (4)



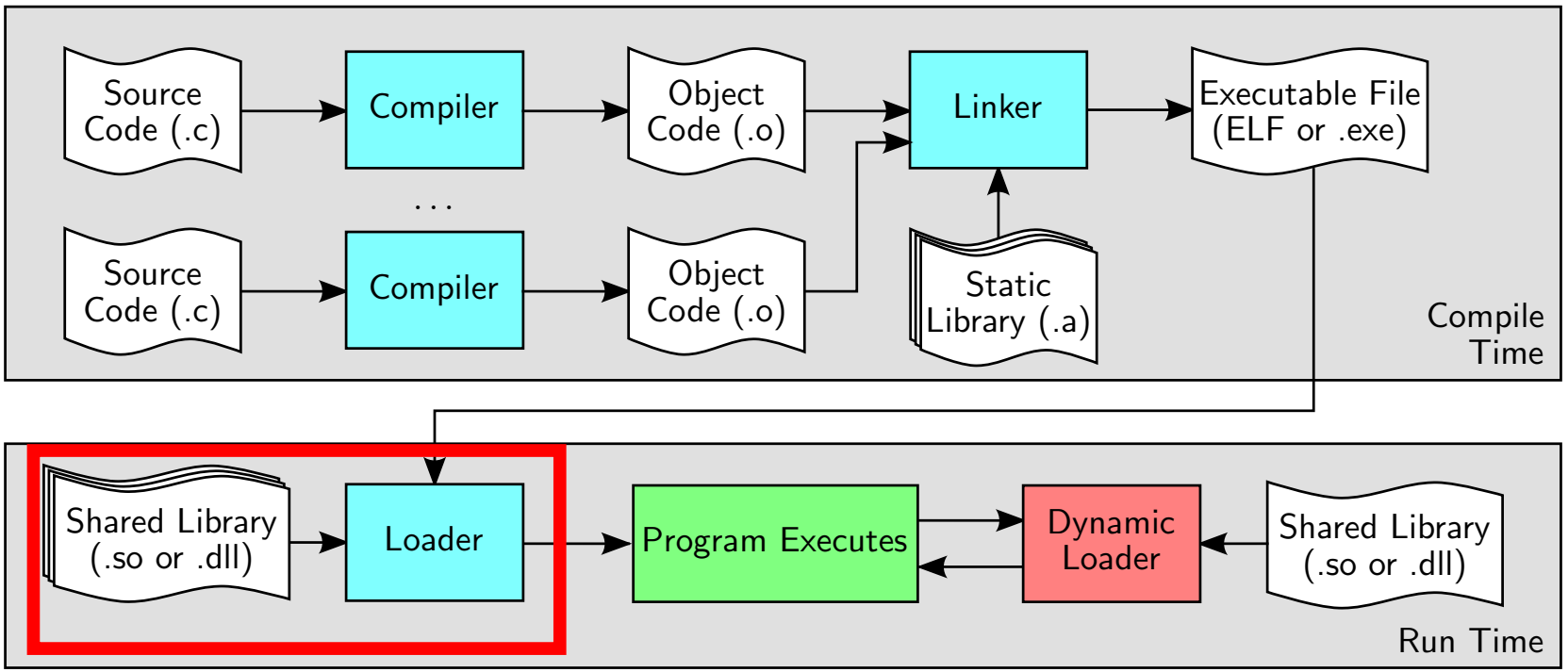
Alternatively, the linker can leave some relocations and symbols undefined in the final executable and plan to resolve the dependencies at run time.

Linking and Loading (5)



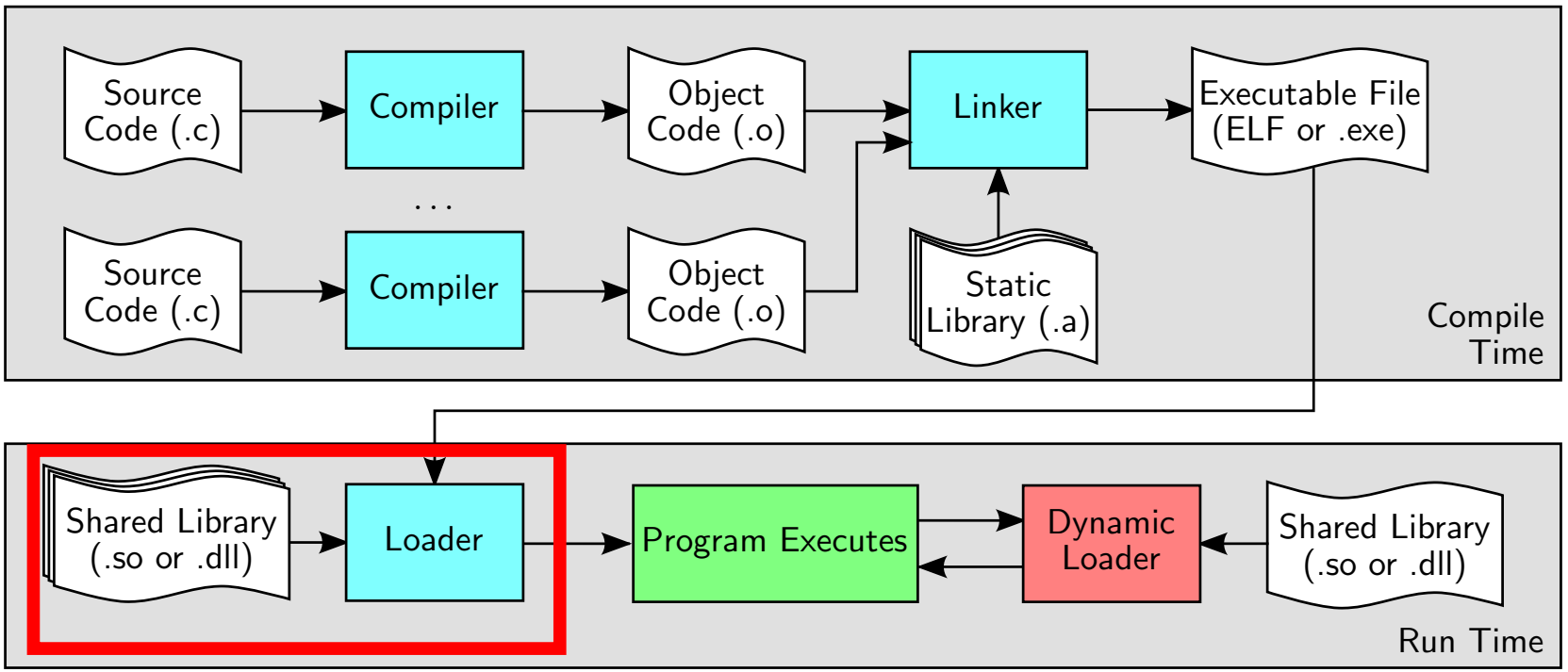
When the program runs, a second linking stage (called a **loader**) will load the executable into memory, along with code or data from a **shared library** (normally with the `.so` extension).

Linking and Loading (6)



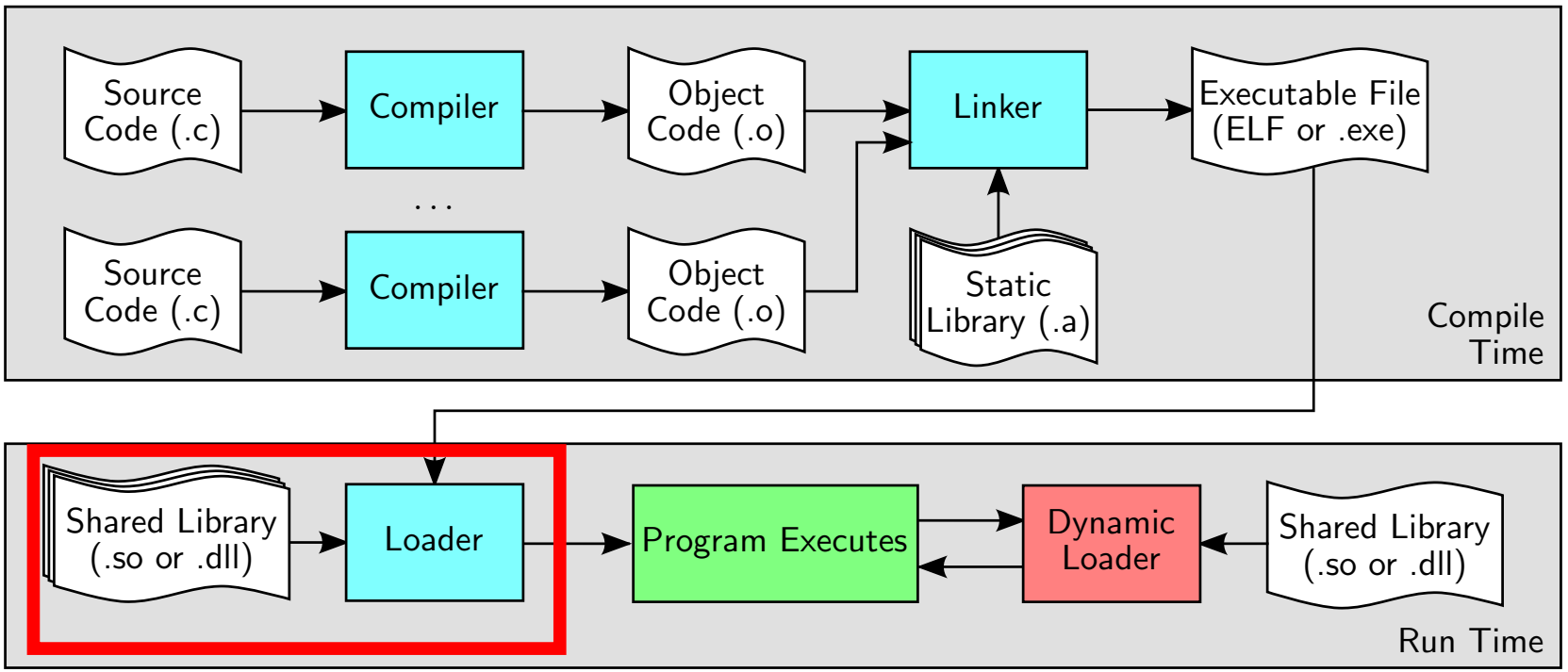
The executable file does not need to contain copies of shared library data. Instead, the linker only needs to add a stub indicating which symbols are needed and which libraries should be used to find them.

Linking and Loading (7)



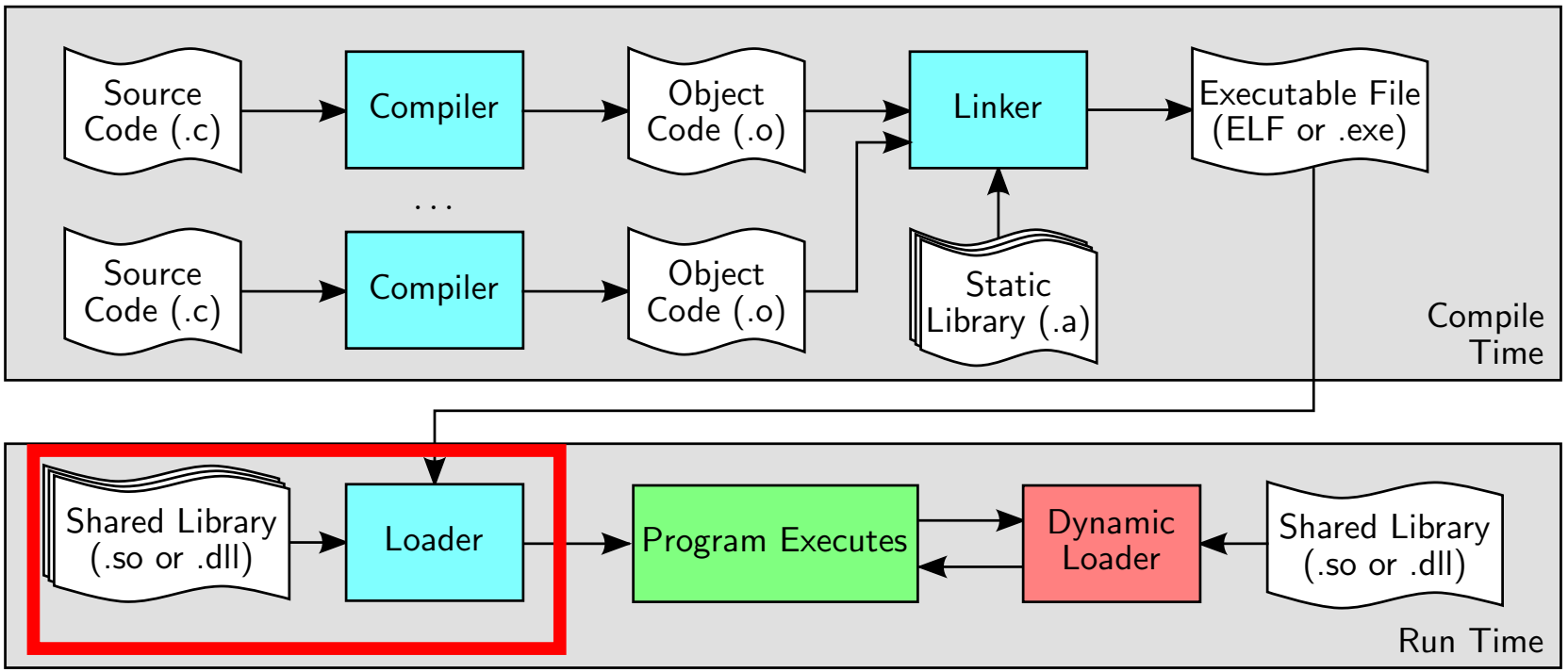
As a result, if a particular library is needed by a large number of programs, it can save space to use a shared library instead of a static library.

Linking and Loading (8)



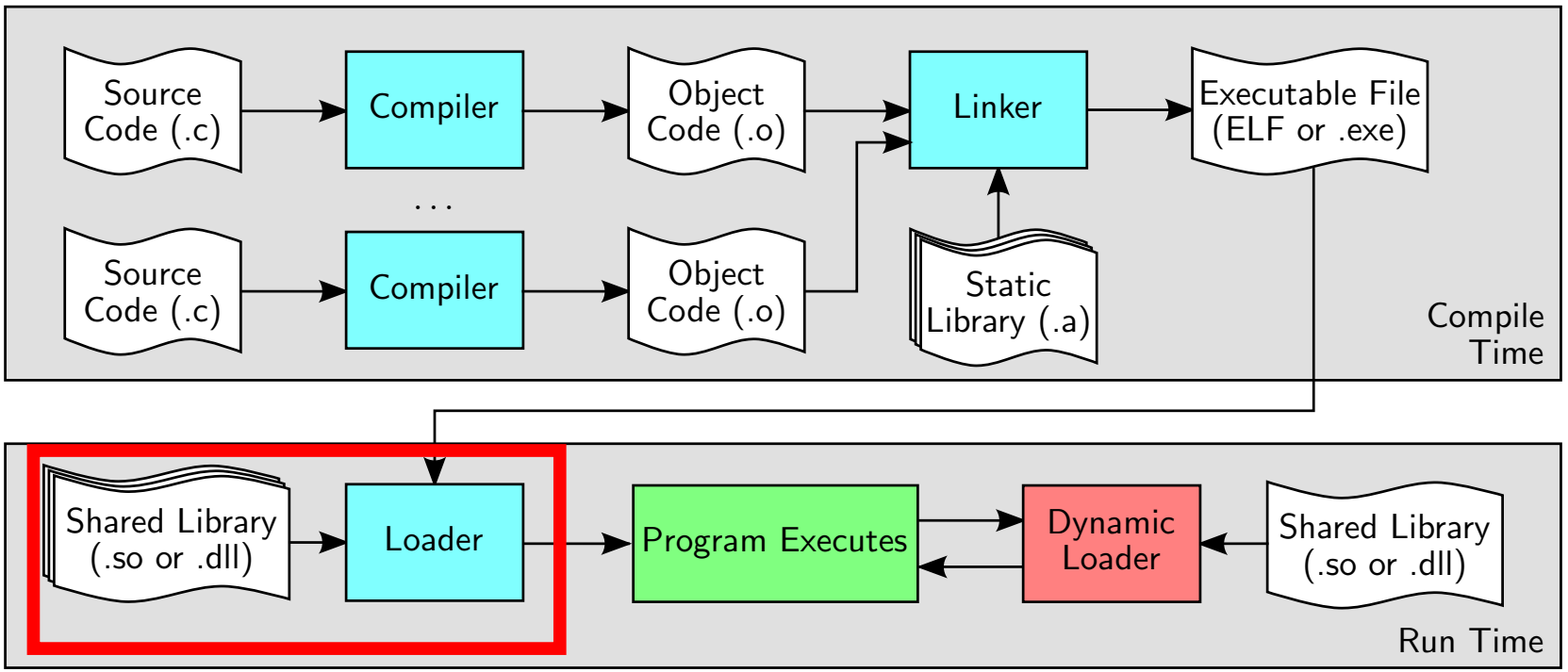
Also, since shared libraries are loaded at run time, a new version of the library can be easily integrated with existing code.

Linking and Loading (9)



This property has some downsides, namely **DLL hell**, which is caused when a program needs several different shared libraries which might have version conflicts with each other.

Linking and Loading (10)



Shared library code is usually compiled to be **position independent**, since it is not known in advance where it will be placed in the program's memory space. Position independent code contains **relative addresses only**.

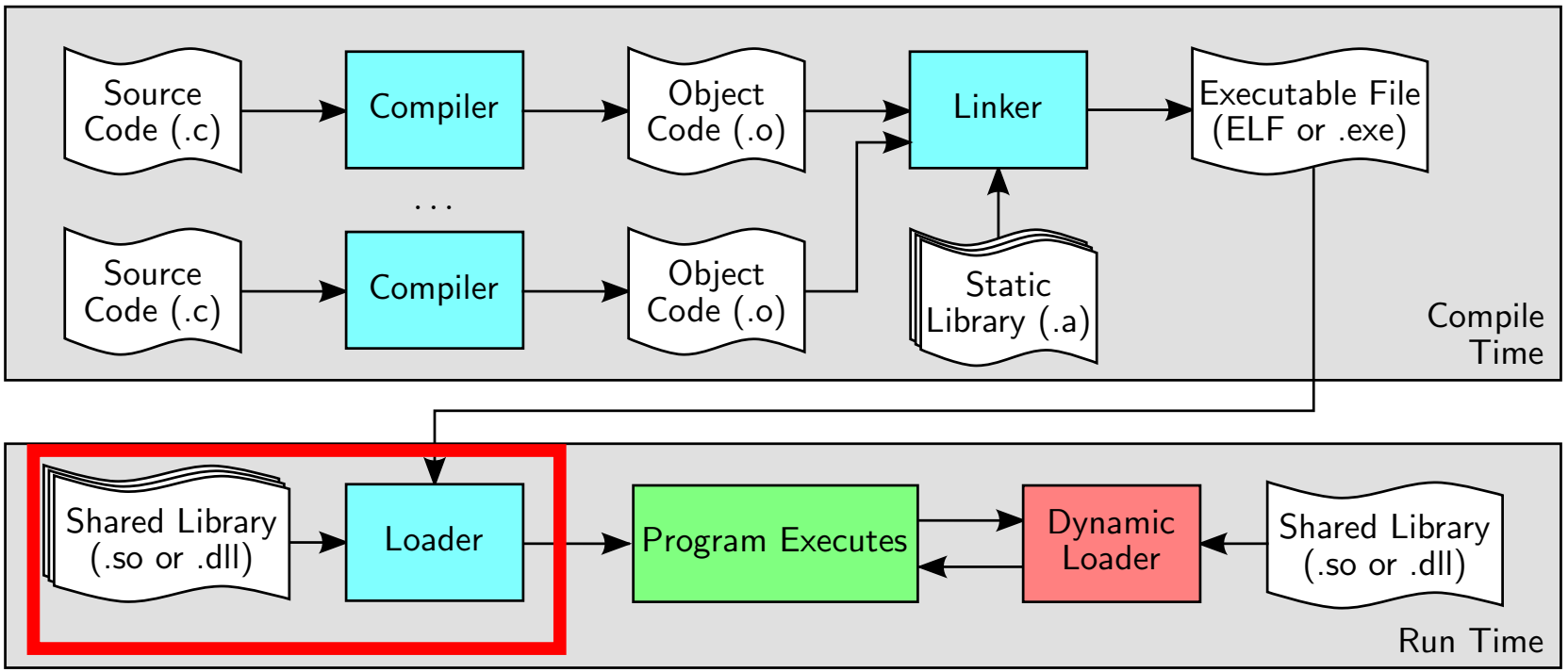
Linking and Loading (11)

It is important to distinguish between relocatable code and position independent code.

Relocatable Code (like `.o` files) uses a relocation table to allow absolute addresses to be set after the initial compilation (usually at linking time). Setting all of the addresses can be time consuming. Using relocatable code allows the final layout decisions to be **deferred**, but not completely ignored (since the addresses still need to be set).

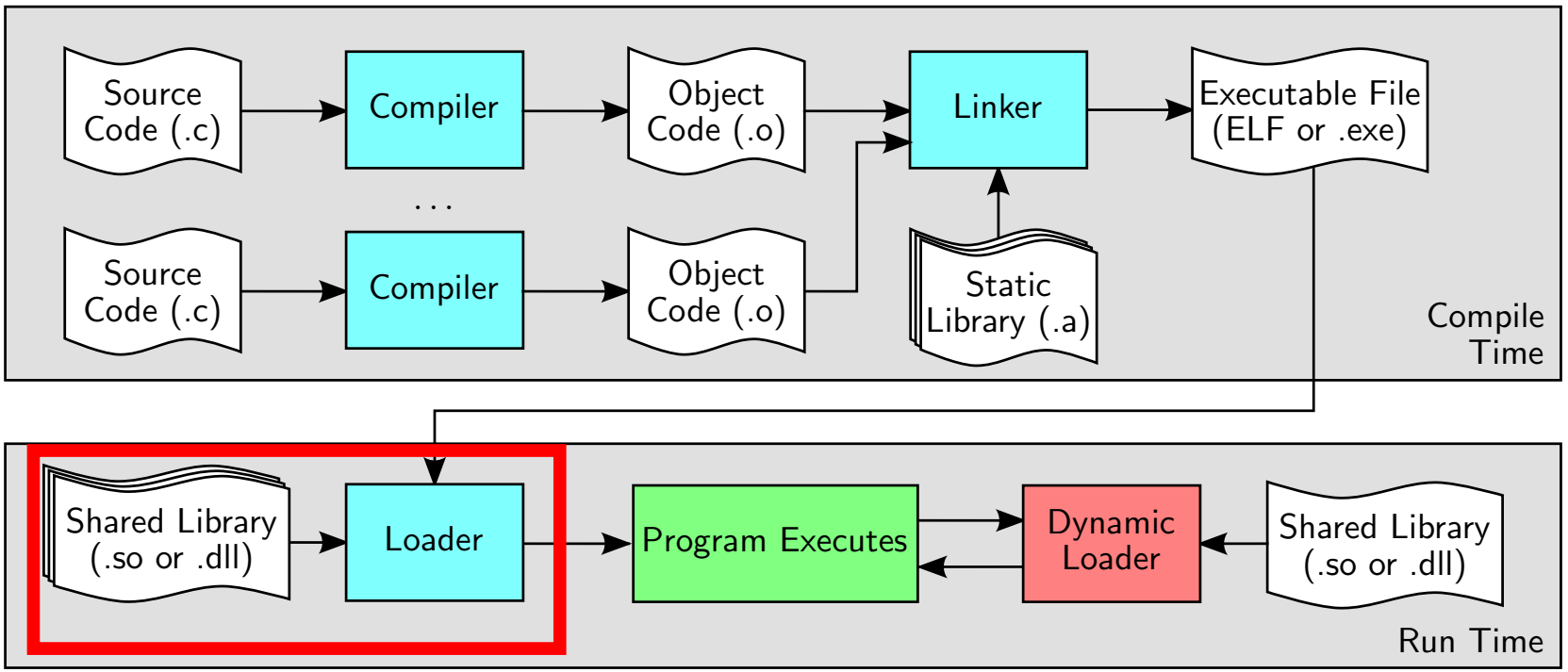
Position Independent Code (PIC) does not use any absolute addresses at all. All load/store instructions use relative addressing, as do all branch instructions. Therefore, position independent code can be moved around memory at will. This allows the exact same code to be mapped into the address spaces of multiple processes in different locations (which is commonly done for shared libraries). It can also be beneficial for security reasons.

Linking and Loading (12)



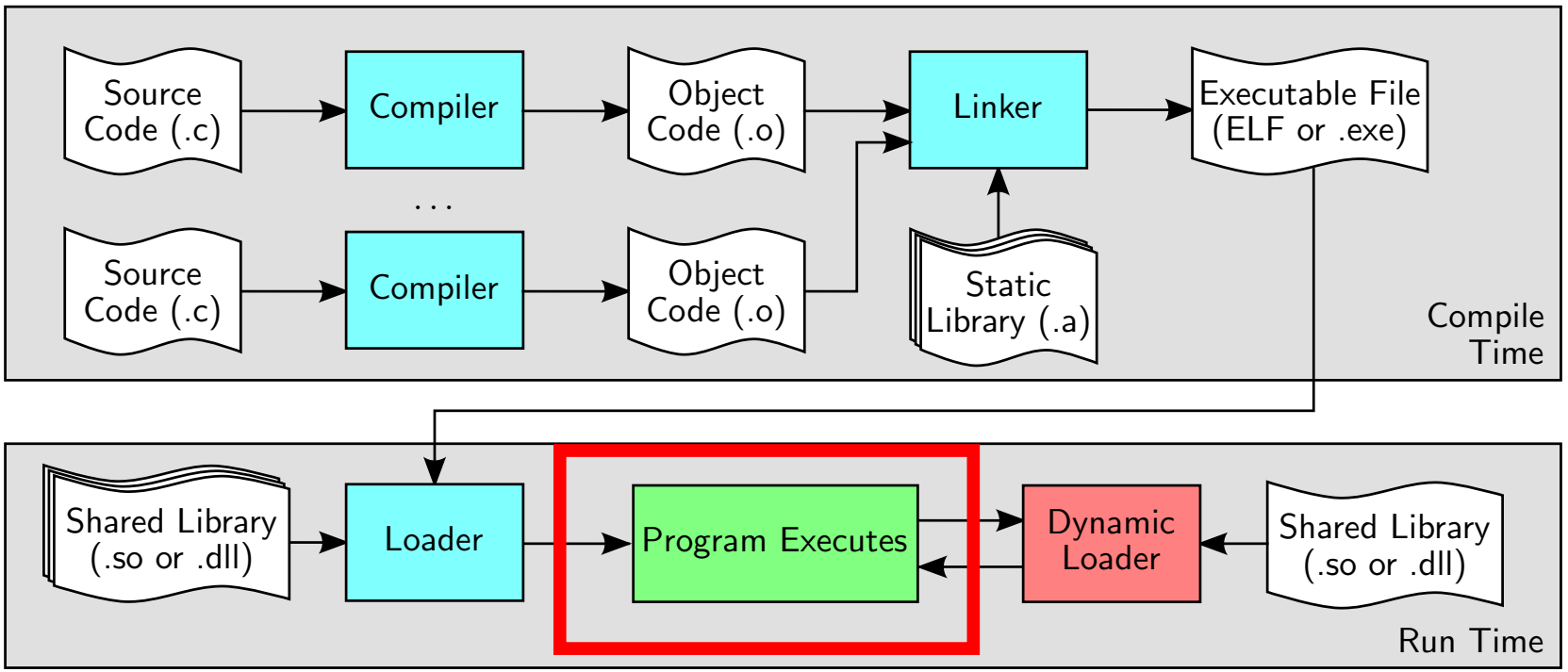
By default, the C build process usually includes the C standard libary (`libc`) as a shared library, since it is a large body of code and is used by many different programs.

Linking and Loading (13)



You can view the set of shared libraries needed by a program using the `ldd` command on Unix-like machines (e.g. '`ldd my_program`')

Linking and Loading (14)



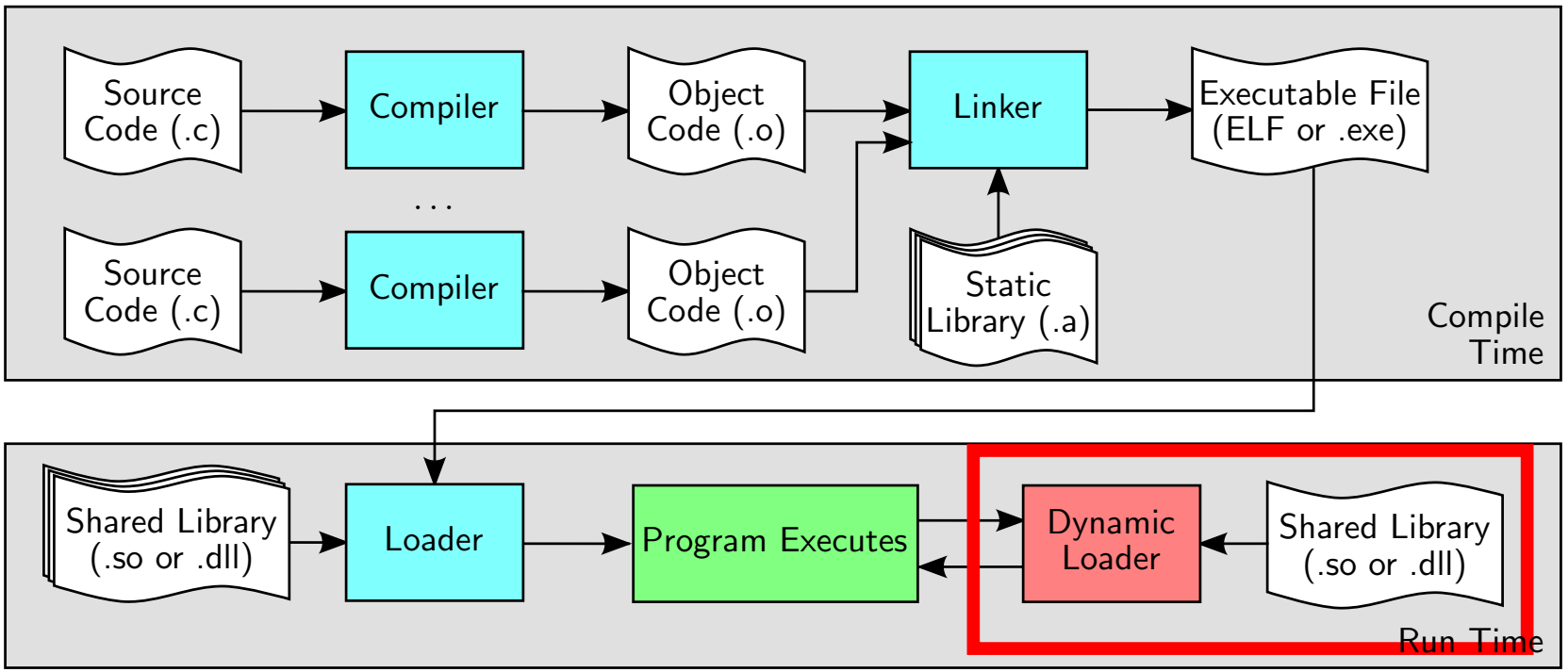
After the load stage is finished, all calls and memory accesses will have valid destination addresses. However, some of the library code may still not be loaded.

Linking and Loading (15)

When a program starts, the loader may immediately load all code from the required shared libraries into memory and set all of the pointers from the program to the library data. Doing so slows down the load process, but ensures that all calls to shared library functions will succeed with no extra overhead.

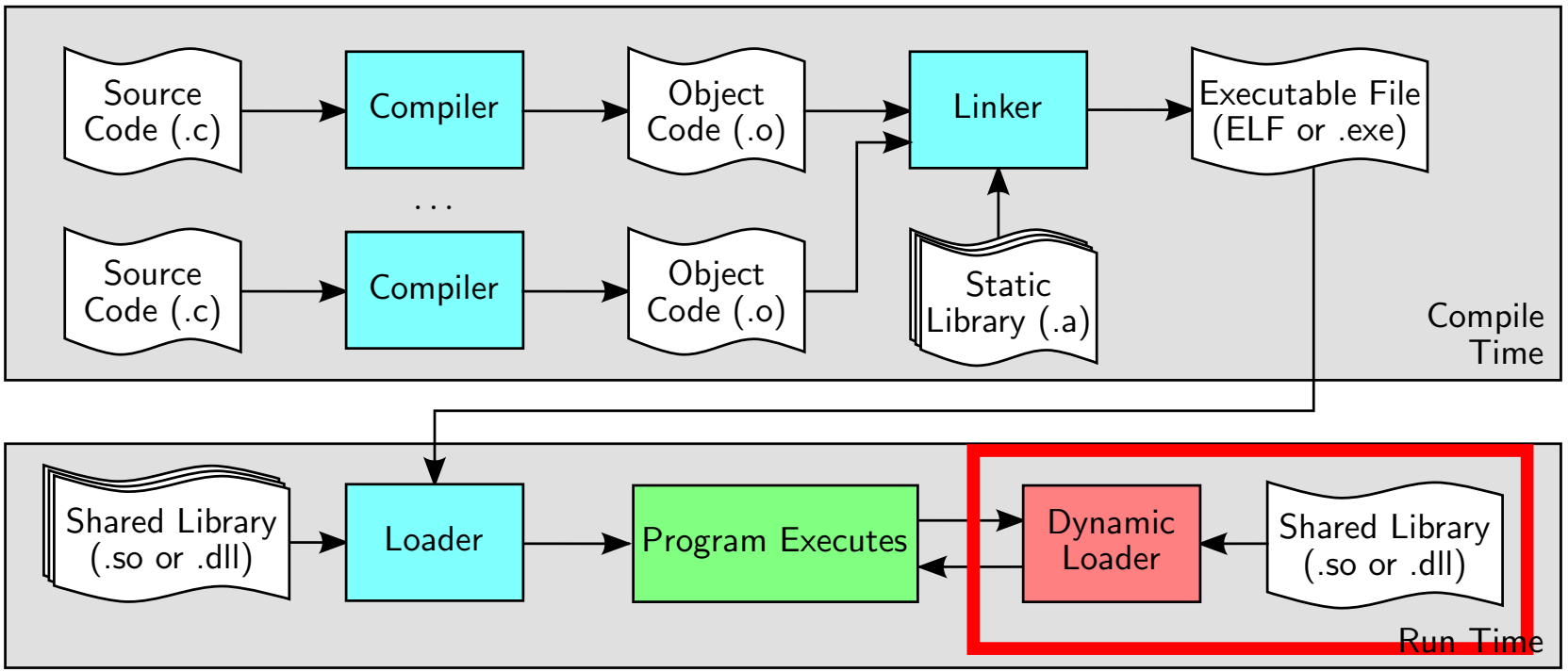
Alternatively, a **lazy** loading strategy can be used: The loader will find and open the shared library files (to ensure that all needed symbols are present), but not actually set the pointers from the main program to the library. Instead, all of the function calls from the main program to the library will go through a **stub function**, which, if it is ever called, will load the necessary library function and set up the pointers accordingly. Any subsequent calls will go directly to the library function.

Linking and Loading (16)



It is possible to use (platform specific) system calls to dynamically load a library during execution. The dynamic load process is far above the level of this course (and requires very advanced and ugly semantics in C).

Linking and Loading (17)



(An example of dynamic loading has been posted on conneX anyway. It is not examinable.)

Linking and Loading (18)

The different ways of binding code dependencies together have different (and sometimes confusing names).

Static Linking refers to linking `.o` files and possibly static libraries (`.a` files) to produce an executable.

Dynamic Linking refers to linking executables against shared libraries (`.so` files) which are then loaded at run time.

Dynamic Loading refers to the program-initiated process of opening shared libraries and finding symbols inside them during the execution of the program.