#1 : Program Execution

Computer Architecture 2021/2022

Ricardo Rocha

Computer Science Department, Faculty of Sciences, University of Porto

Translating and Starting a Program

We can consider **four hierarchical steps** when transforming a C program in a file on disk into a process running on a computer:

- **Compiler** transforms the high-level language program to an assembly language program, a symbolic form of what the machine understands

- **Assembler** turns the assembly language program into an object file, which includes machine code, data, and information needed to execute the program

- **Linker** combines independently object files and resolves all undefined labels into an executable file

- **Loader** places an executable file in main memory so that it is ready to execute
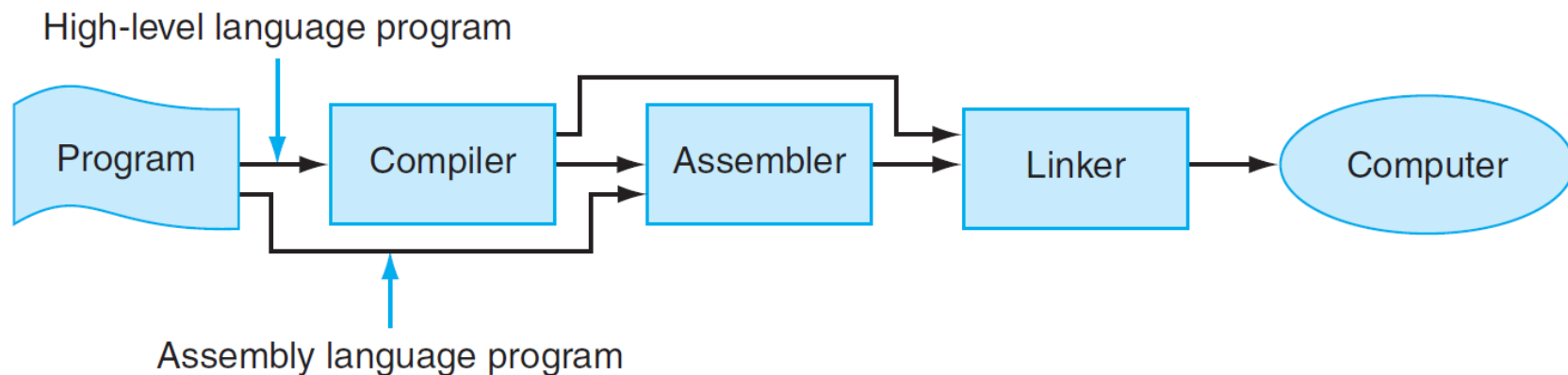
Object Files

The object file for UNIX systems typically contains six distinct pieces:

- **Object file header** describes the size and position of the other 5 pieces

- **Text segment** contains the machine language code

- **Static data segment** contains data allocated for the life of the program

- **Relocation information** identifies instructions and data words that depend on absolute addresses when the program is loaded into memory

- **Symbol table** contains the remaining labels that are not defined, such as global definitions and external references

- **Debugging information** contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

Linking Object Files

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels.

- Such references occur in branch/jump instructions and data addresses

- The linker produces an executable file that has the same format as an object file, except that it contains no unresolved references or relocation information
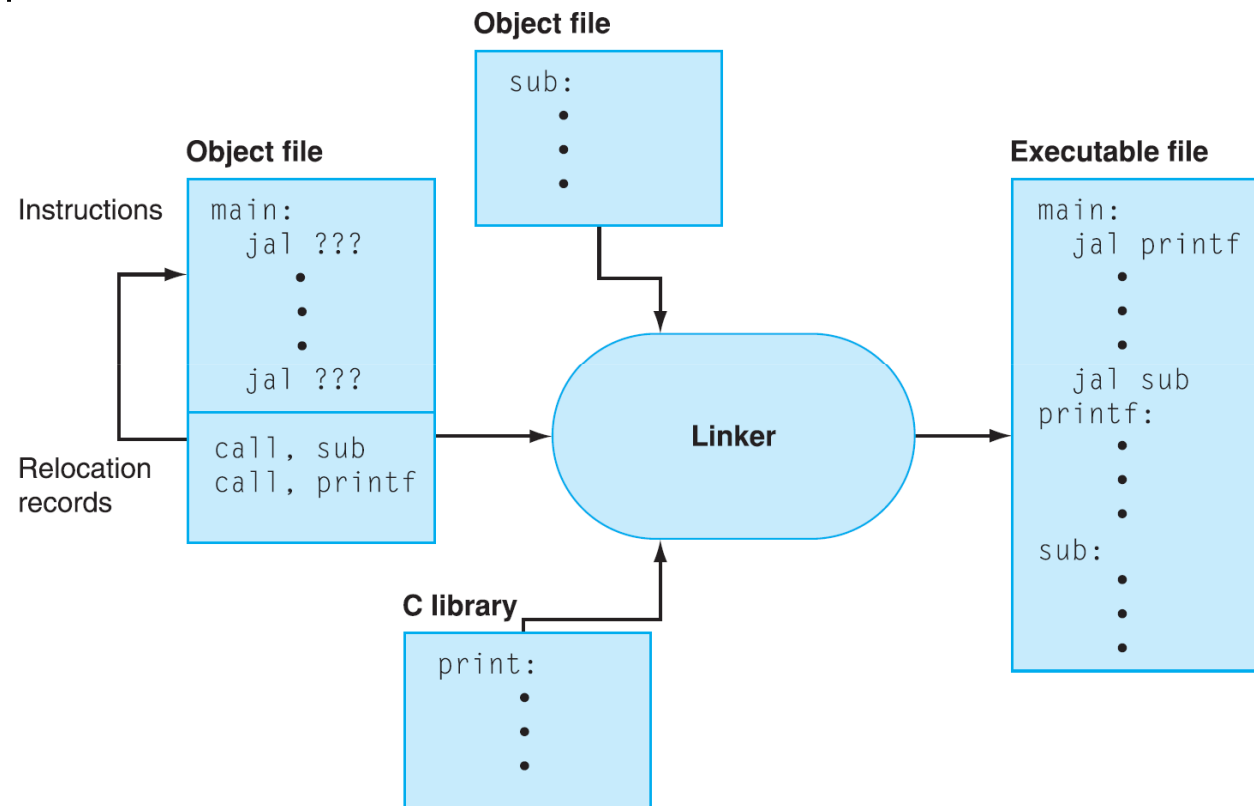
Dynamic Linking

Static linking has a few disadvantages:

- The library becomes part of the executable code and, if a new library version is released, the statically linked program keeps using the old version
- It loads all routines in the library even if those calls are not executed

These disadvantages lead to **dynamically linked libraries (DLLs)** where **each library routine is loaded only when it is needed**:

- Initially, the main routine is loaded into memory and executed
- When a routine calls another routine, the calling routine first checks to see whether the other routine has been loaded and, if not, a **relocatable linking loader** is called to load the desired routine into memory
- A **stub** is included in the binary program for each library routine reference that indicates how to locate the appropriate library routine and load it
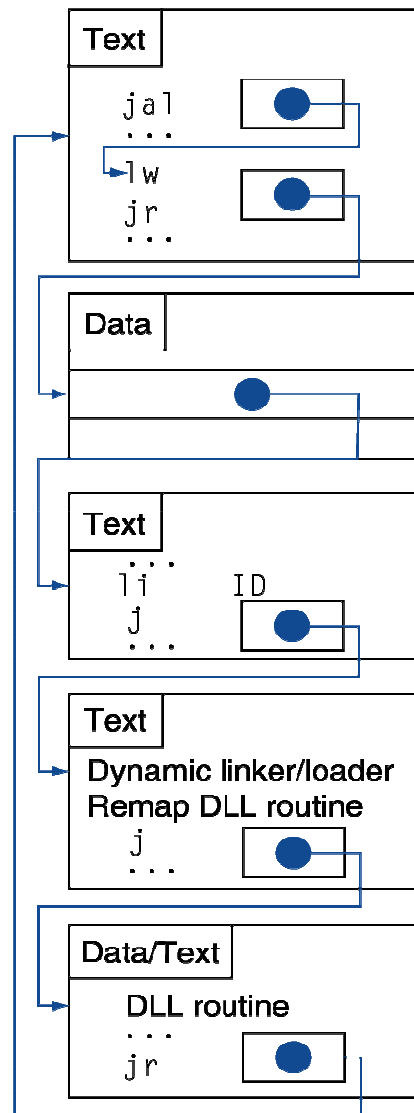
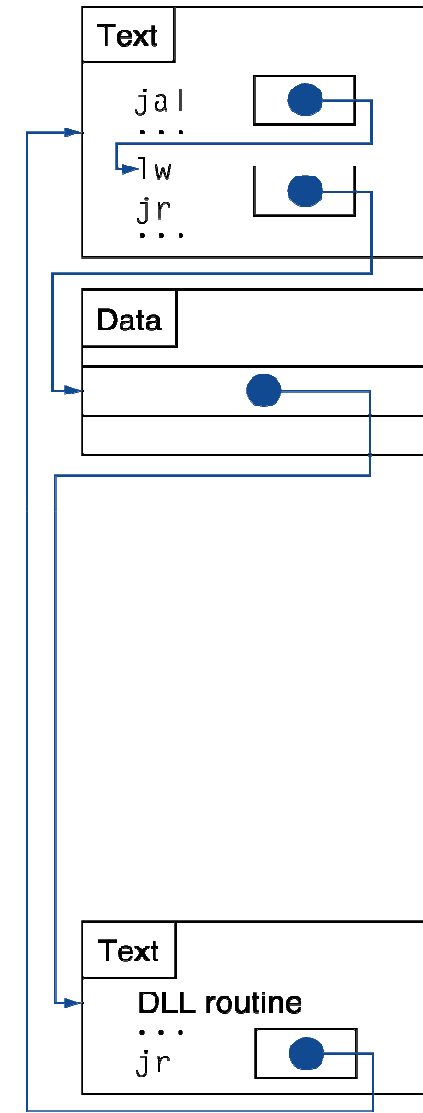# Dynamic Linking

Indirection table

Stub: loads routine ID and jumps to linker/loader

Linker/loader code

DLL code

a. First call to DLL routine
b. Subsequent calls to DLL routine