#2 : Program Execution

Computer Architecture 2020/2021

Ricardo Rocha

Computer Science Department, Faculty of Sciences, University of Porto

Slides based on the book

'Computer Organization and Design, The Hardware/Software Interface, 5th Edition

David Patterson and John Hennessy, Morgan Kaufmann'

Sections 2.12 and A.1 – A.4

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 step**
- **Assembler step**
- **Linker step**
- **Loader step**

Some systems combine these steps to reduce translation time, but these are the logical four steps that programs go through.

Translating and Starting a Program

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

The **assembler** turns the assembly language program into an object file, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

The **linker** combines independently assembled object files and resolves all undefined labels into an executable file.

The **loader** places an executable file in main memory so that it is ready to execute.

Translating and Starting a Program

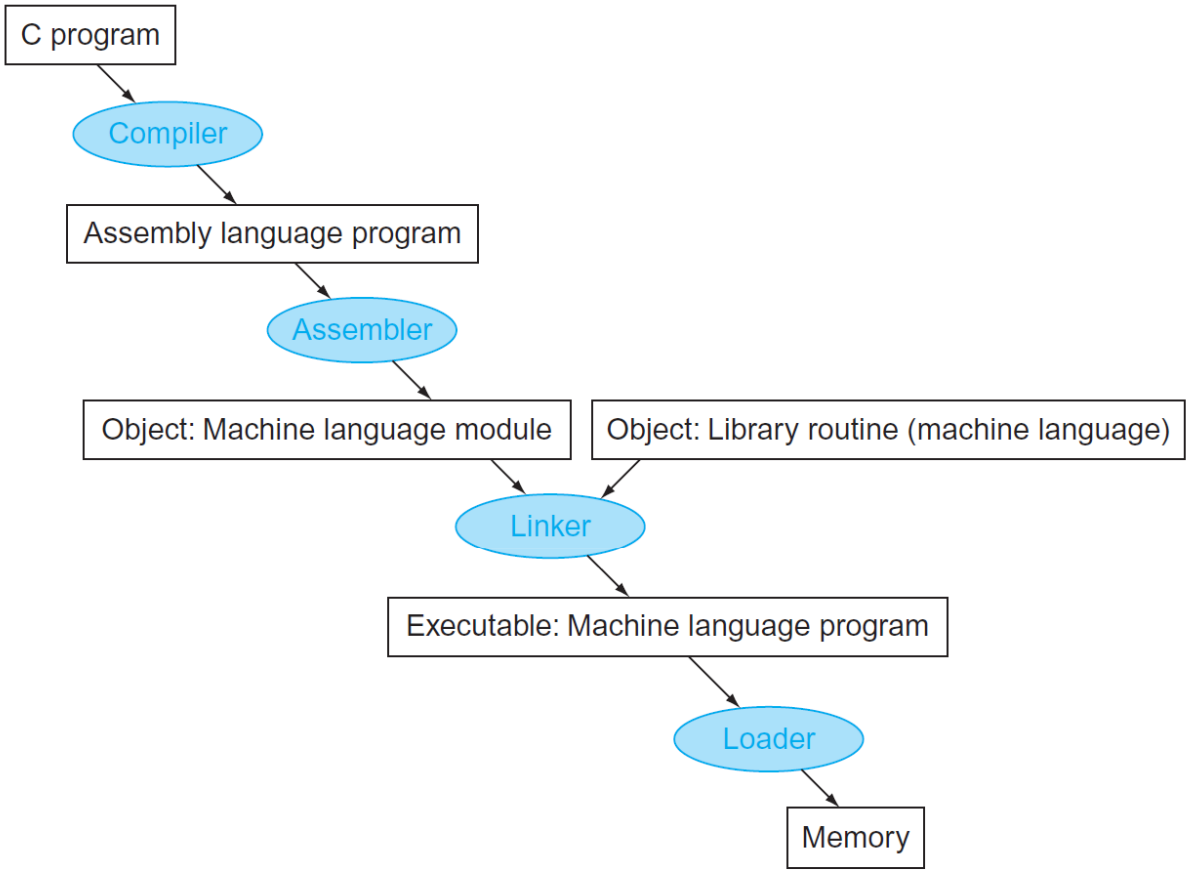


FIGURE 2.21 A translation hierarchy for C. A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named `x.c`, assembly files are `x.s`, object files are named `x.o`, statically linked library routines are `x.a`, dynamically linked library routes are `x.so`, and executable files by default are called `a.out`. MS-DOS uses the suffixes `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL`, and `.EXE` to the same effect.

Translating and Starting a Program

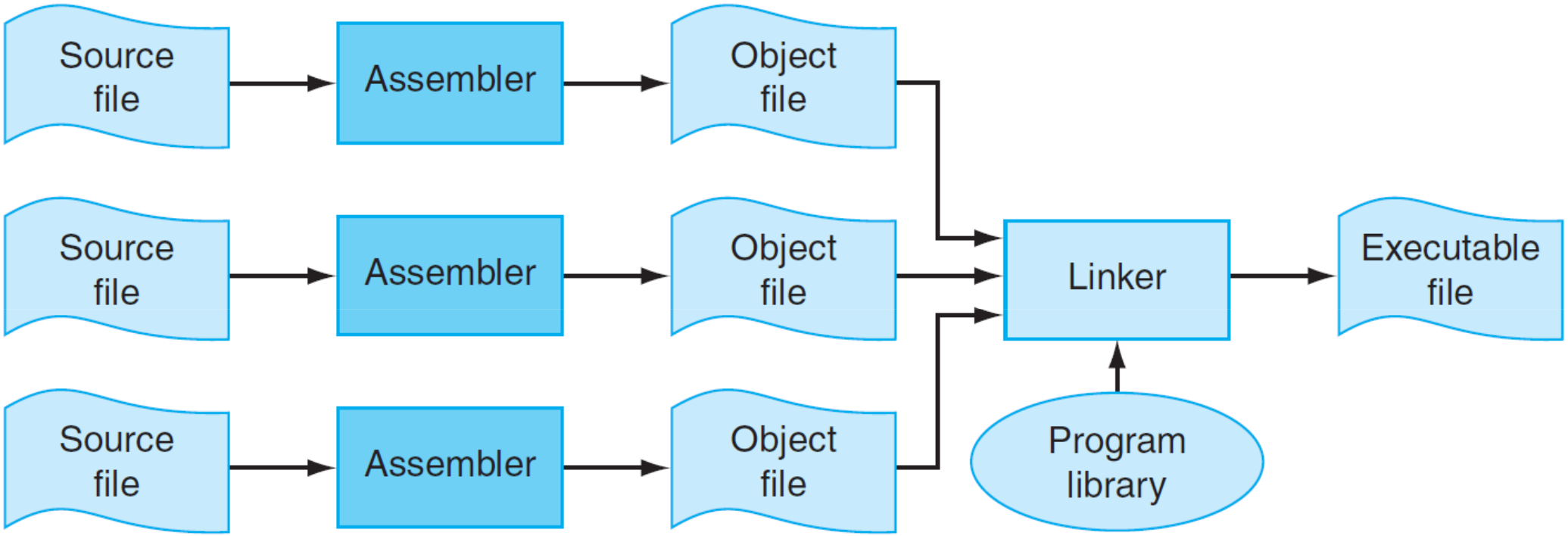


FIGURE A.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Assembly Language

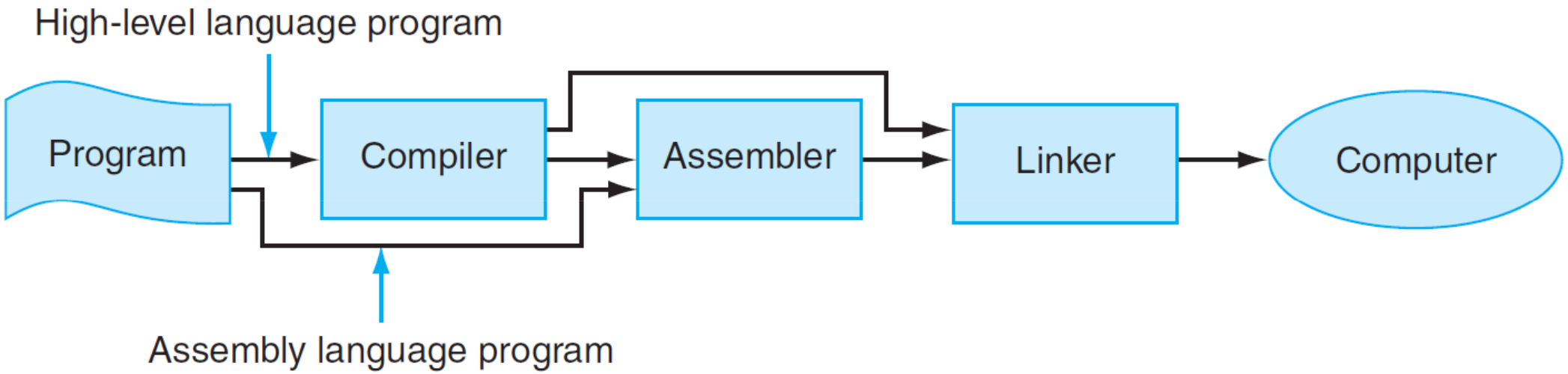


FIGURE A.1.6 Assembly language either is written by a programmer or is the output of a compiler.

From C to Machine Code

```
   #include <stdio.h>

int
main (int argc, char *argv[])
{
     int i;
     int sum = 0;

     for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
     printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

FIGURE A.1.5 The routine in Figure A.1.2 written in the C programming language.

High-level C program: code is short and clear – variables have mnemonic names and the loop is explicit rather than constructed with branches.

From C to Machine Code

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

Assembly program I: code more difficult to follow, because many simple operations are required to accomplish simple tasks and because assembly language's lack of control flow constructs provides few hints about the program's operation.

FIGURE A.1.4 The same routine as in Figure A.1.2 written in assembly language with labels, but no comments. The commands that start with periods are assembler directives (see pages A-47–49). `.text` indicates that succeeding lines contain instructions. `.data` indicates that they contain data. `.align n` indicates that the items on the succeeding lines should be aligned on a 2^n byte boundary. Hence, `.align 2` means the next item should be on a word boundary. `.globl main` declares that `main` is a global symbol that should be visible to code stored in other files. Finally, `.asciiz` stores a null-terminated string in memory.

From C to Machine Code

```
addiu     $29, $29, -32
sw        $31, 20($29)
sw        $4, 32($29)
sw        $5, 36($29)
sw        $0, 24($29)
sw        $0, 28($29)
lw        $14, 28($29)
lw        $24, 24($29)
multu     $14, $14
addiu     $8, $14, 1
slti      $1, $8, 101
sw $8,    28($29)
mflo      $15
addu      $25, $24, $15
bne       $1, $0, -9
sw        $25, 24($29)
lui       $4, 4096
lw        $5, 24($29)
jal       1048812
addiu     $4, $4, 1072
lw        $31, 20($29)
addiu     $29, $29, 32
jr        $31
move      $2, $0
```

Assembly program II: code even more difficult to follow because memory locations are named by their address rather than by a symbolic label.

FIGURE A.1.3 The same routine as in Figure A.1.2 written in assembly language. However, the code for the routine does not label registers or memory locations or include comments.

From C to Machine Code

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000001111000000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

Machine code: with considerable effort, we could use the opcode and instruction format tables to translate the instructions into a symbolic program similar to the previous one.

FIGURE A.1.2 MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100.

Address Binding

Addresses are represented in different ways at different stages of a program's life:

- Source code addresses are usually symbolic (e.g., **variable xpto**)
- Compiler/assembler binds symbolic addresses to relocatable addresses (e.g., **604 bytes from the beginning of this module**)
- Linker/loader binds relocatable addresses to absolute addresses (e.g., **address 0x0FFF0904**)
- Each binding maps one address space into another

Object File

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

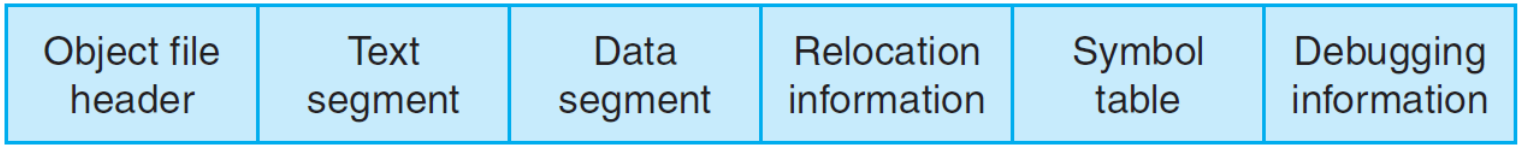


FIGURE A.2.1 Object file. A UNIX assembler produces an object file with six distinct sections.

Linking Object Files

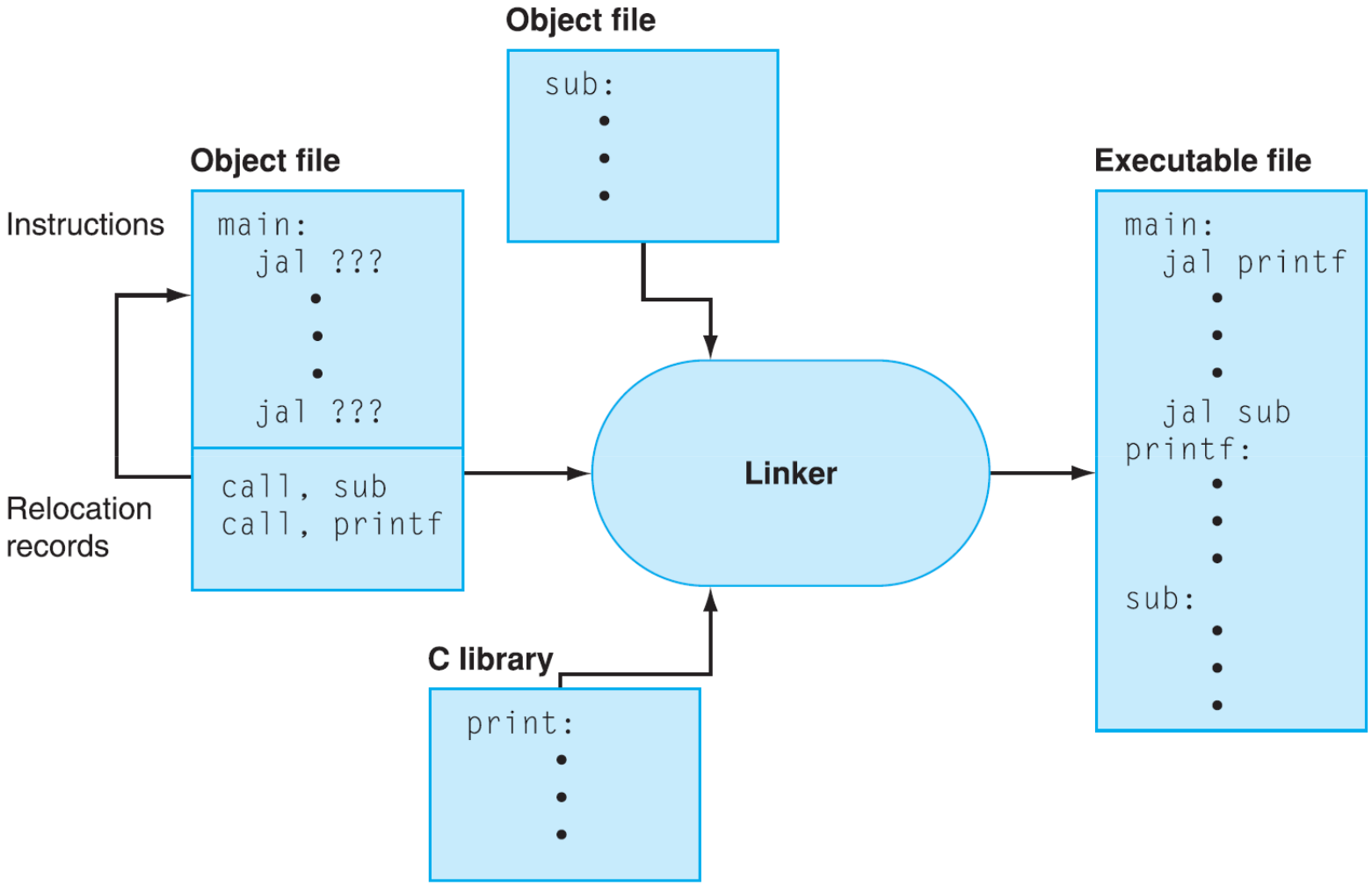


FIGURE A.3.1 The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

Linking Object Files

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.

The linker typically includes three steps:

- **Find library routines used by the program**
- **Merge segments** by placing code and data modules symbolically in memory and **relocate its instructions** by adjusting absolute references
- **Resolve references among files**

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 instructions, jump instructions, and data addresses, so the job of this program is much like that of an editor – it finds the old addresses and replaces them with the new addresses
- Could leave location dependencies for fixing by a relocating loader, but with virtual memory there is no need to do this since a program can be loaded into absolute location in virtual memory space

Loading a Program

The loader typically includes six steps to load an executable file into memory:

- Read file header to determine size of the text and data segments
- Create address space large enough for the text and data segments
- Copy the instructions and data from the executable file into memory (or set page table entries so they can be faulted in)
- Copy the program's arguments (if any) onto the stack
- Initialize the machine registers and set the stack pointer to the top of the stack
- Jump to startup routine, which copies the program's arguments into the argument registers and calls the main routine of the program. When the main routine returns, the startup routine terminates the program with an exit() system call

Dynamic Linking

Although static linking is the fastest way to call library routines, it has a few disadvantages:

- The library routines become part of the executable code (if a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version)
- It loads all routines in the library that are called anywhere in the executable, even if those calls are not executed (the library can be large relative to the program – for example, the standard C library is 2.5 MB)

These disadvantages lead to **dynamically linked libraries (DLLs)**, where each library routine is linked and loaded only after it is called, i.e., **linking is postponed until execution time**.

- Requires procedure code to be relocatable
- Automatically picks up new library versions and avoids linking all referenced libraries/routines

Dynamic Linking

With dynamic linking, a **routine is loaded only when it is needed**:

- All routines are kept on disk in a relocatable load format
- 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, the 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

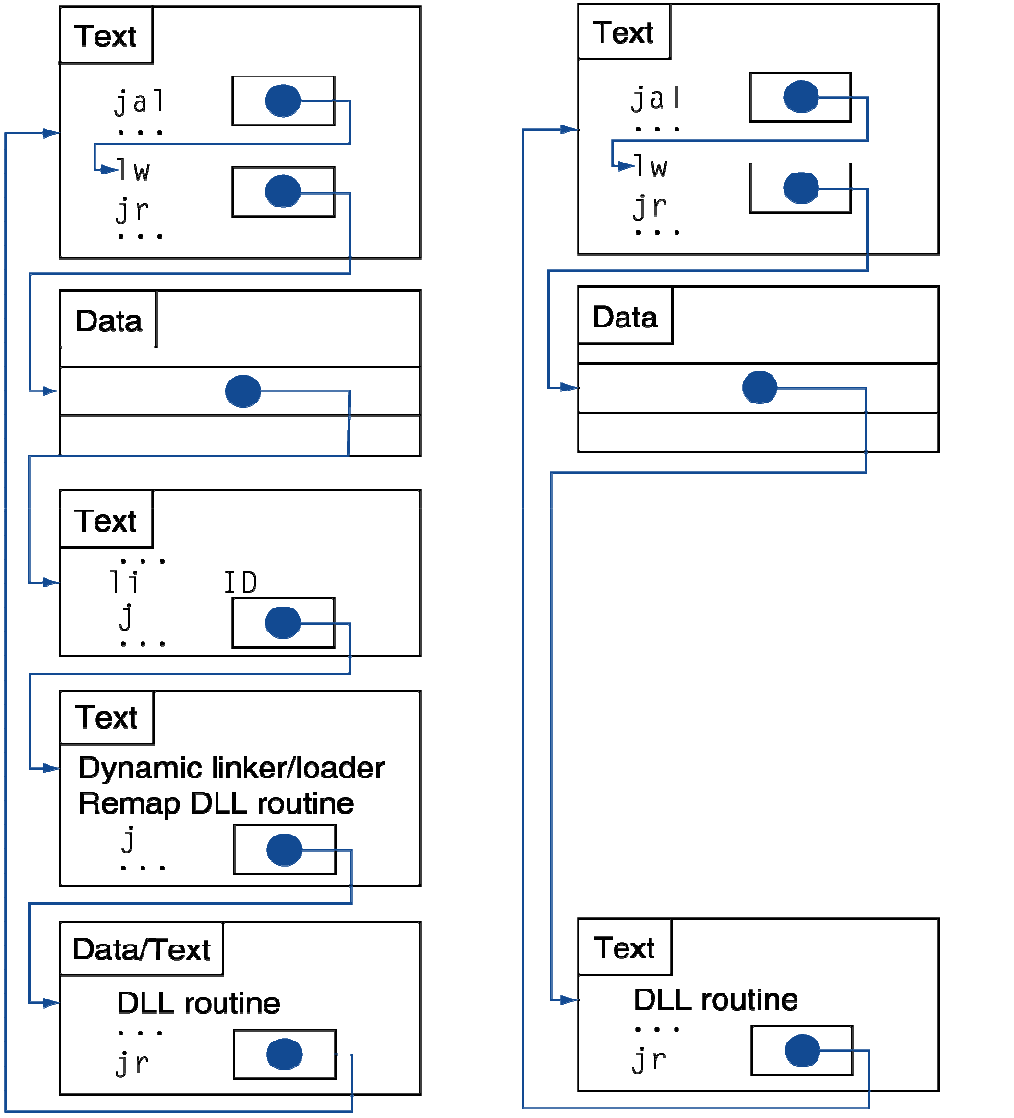
Dynamic Linking

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

Dynamic Linking

Dynamic linking allows for a **better memory-space utilization**

- Although the total program size may be large, the portion that is used (and hence loaded) may be much smaller
- Particularly useful for system libraries (without this facility, each program must include a copy of the library in the executable image) and when large amounts of code handle infrequently occurring cases (such as error routines)
- Processes that use the same library execute only one copy of the library code