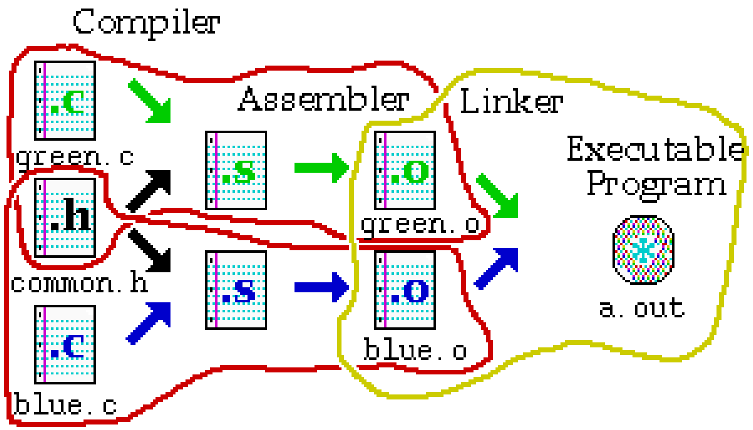
# ساختار و زبان کامپیوتر

## فصل پنجم

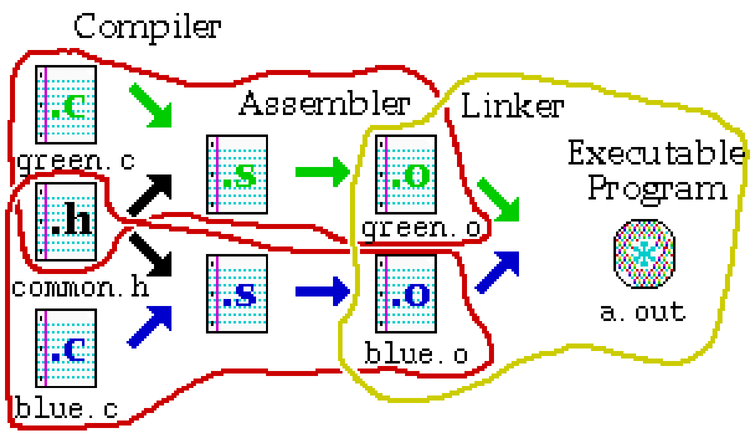
## ترجمه و راه‌اندازی برنامه‌ها



# Computer Structure and Machine Language

## Chapter Five

## Translating & Starting a Program



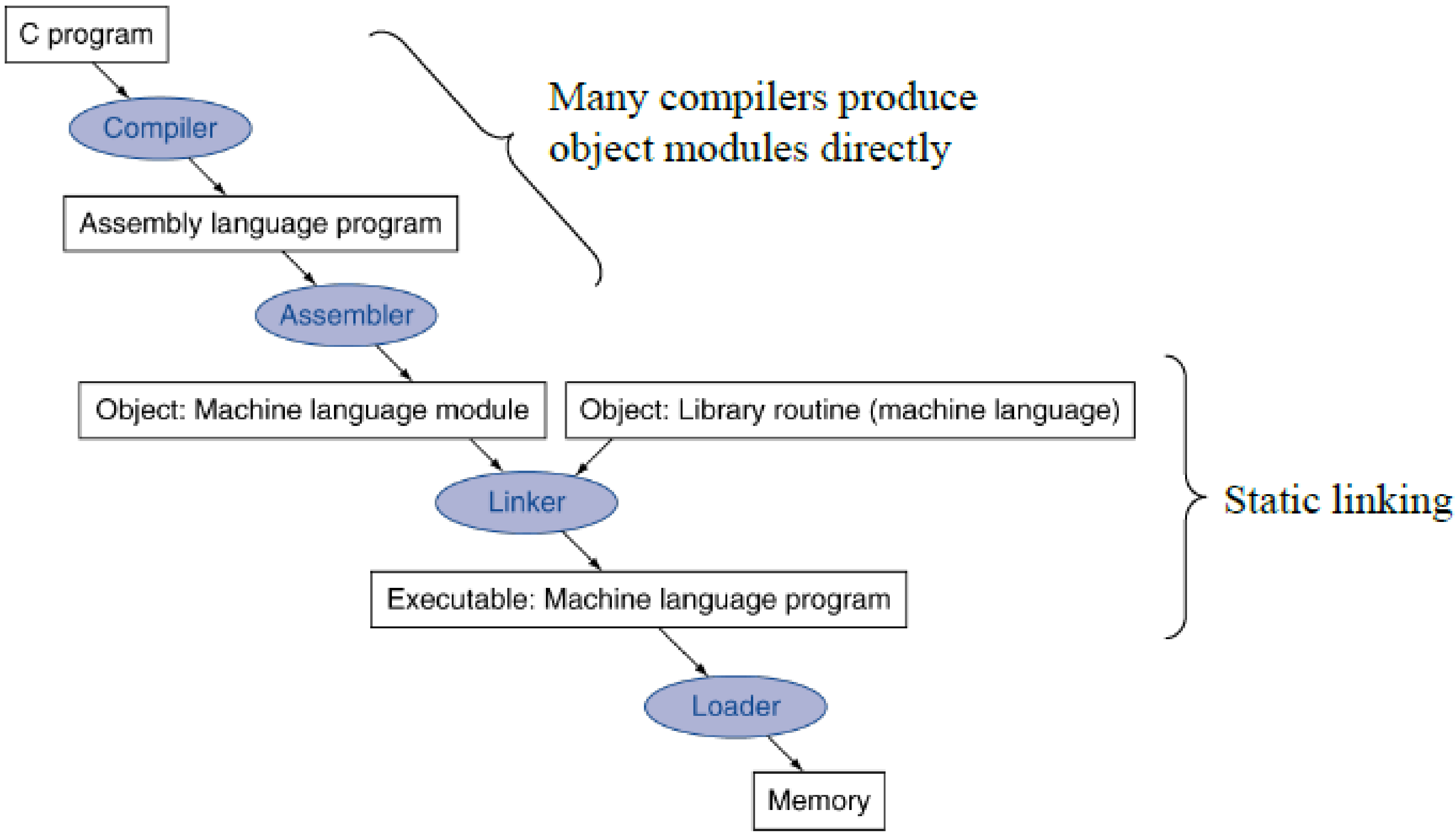
# Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- D. Patterson & J. Hennessey, "Computer Organization & Design, The Hardware/Software Interface", 6th Ed., MK publishing, 2020
- A. Tanenbaum, "Structured Computer Organization", 5th Ed., Pearson, 2006

# Contents

# A Translation Hierarchy for C



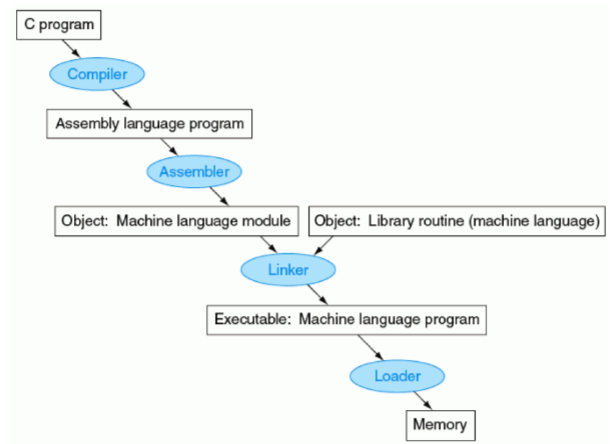
# UNIX / DOS-Win File Extensions

- Unix / DOS-Windows
  - C source files: x.c / x.c
  - Assembly files: x.s / x.asm
  - Object files: x.o / x.obj
  - Statically linked library routines: x.a / x.lib
  - Dynamically linked library routines: x.so/x.dll
  - Executable files: A.out/A.exe

# Contents

# Compilers

- Translates a C Program into an Assembly Program
- Input
  - High-level language code
    - e.g., C, Pascal, etc.
- Output
  - Assembly language code
    - e.g., MIPS assembly code
  - Still different from object code (machine language)
- Some compilers produce object code directly
  - A matter of compilation speed vs. compiler simplicity



# Contents

- Introduction
- Compilers
- Assemblers
- Linkers
- Loaders
- Translation vs. Interpretation
- Java Applications

# Assembly vs. Machine Language

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

MIPS machine language

MIPS assembly language

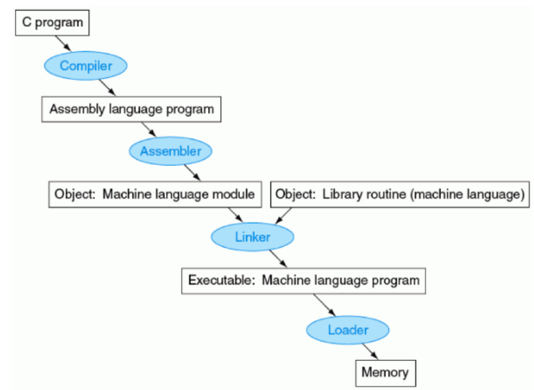
```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        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"
```

Same routine in C

```
#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);
}
```

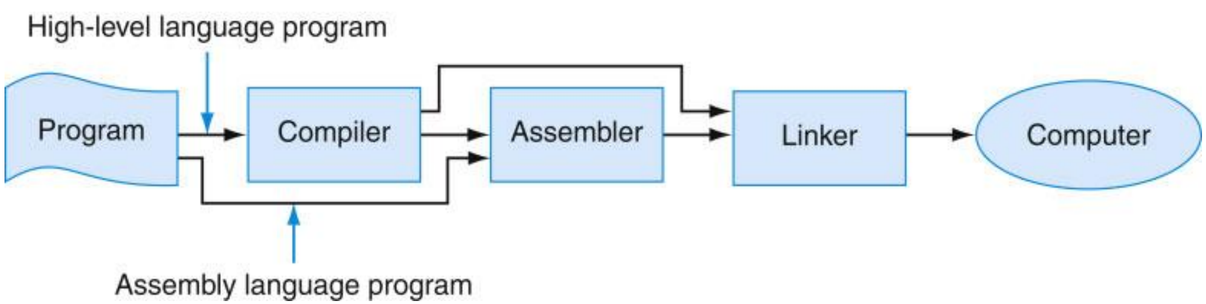
# Assembler

- Translate assembly program to binary code
- Input
  - Assembly language code
    - e.g., `foo.s` for MIPS
- Output
  - Object code (machine language)
    - Produced machine language
    - e.g., `foo.o` for MIPS
  - Information tables



# Why Assembly? (40 years ago)

- Although compilers were available 30 years ago, many programs were written in assembly language. Why?
  - RAM sizes were small
  - Code density was a big concern
  - Compilers were inefficient



# Why Assembly? (now)

- Assembly language is still used to write programs
  - where speed or size is critical
  - where there is no high-level language available
  - to exploit hardware features that have no analogues in high-level languages
  - to exploit specialized instructions (string copy, pattern matching...)
- Hybrid approach: Most of the program is written in a high-level language while time-critical sections are written in Assembly

# Assembly Language Drawbacks

- Programs written in assembly language are inherently machine-specific and must be totally rewritten to run on another computer architecture
- Assembly language programs are longer than the equivalent programs written in a high-level language
    - lessens programmers' productivity
    - contain more bugs
- Assembly programs are usually hard to read, because of their lack of structure (e.g. if-then & loops)

# Assembler Steps

- Read and use directives
- Replace pseudo-instructions
- Replace macros
- Produce machine language
- Creates object file

# Assembler Directives

- Give directions to Assembler, but do not produce machine instructions
  - **.text:** Subsequent items put in user text segment (machine code)
  - **.data:** Subsequent items put in user data segment (binary representation of data in src file)
  - **.globl sym:** declares global symbol **sym** that can be referenced from other files
  - **.asciiz str:** Store string **str** in memory and null-terminate it

# Pseudo-instructions

- Instructions provided by an assembler but not implemented in hardware
  - Unlike most assembler instructions that represent machine instructions one-to-one
  - MIPS Examples:

```
move $t0, $t1        →   add $t0, $zero, $t1
blt $t0, $t1, L      →   slt $at, $t0, $t1
                         bne $at, $zero, L
```

assembler temporary register

# Macro

- A pattern-matching and replacement facility
- Provides a mechanism to name frequently used sequence of instructions
- The assembler replaces the macro call with a sequence of instructions
- After replacement the resulting assembly has no sign of the macro
- Permits a programmer to create and name a new abstraction for a common operation (like subroutines)
- Does not cause call and return (unlike subroutines)

# Produce Machine Language

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info within instruction already
- Data/ Code Labels
  - Need to know the absolute addresses
- PC-relative branch
  - once pseudo-instructions are replaced, we know by how many instructions to branch

# “Forward Reference” Problem

- Branch instructions can refer to labels that “forward” in program:

```
        or   $v0, $0, $0
L1:     slt  $t0, $0, $a1
        beq  $t0, $0, L2
        addi $a1, $a1, -1
        j    L1
L2:     add  $t1, $a0, $a1
```

- Solved by taking 2 passes over program
  - 1st pass remembers position of labels
  - 2nd pass uses label positions to generate code

# BackPatching

- Another solution to forward references:
- The assembler builds a (possibly incomplete) binary representation of every instruction in one pass over a program
- Records the undefined label and instruction in a table
- Corrects the binary representation of instructions that contain a forward reference, when the label is defined

# BackPatching (cont.)

- ☺ The assembler only reads its input once
  - ➔ Speeds assembly
- ☹ Requires to hold the entire binary representation in memory
  - ➔ limits the size of programs that can be assembled
- ☹ With several types of branches (various lengths)
  - either Use the largest possible branch
  - or Risk having to go back & readjust instructions to make room for a larger branch

# Absolute Addresses

- What about unconditional jumps? (`j`, `jal`)
  - Jumps require absolute address
  - So, forward or not, still can't generate machine instruction without knowing position of instructions in memory
- What about references to data?
  - Requires full 32-bit address of data
- Can't be determined yet ➔ Need tables

# Relocation Table

- List of "items" this file needs their absolute addresses
- What are they?
  - Any label jumped to
    - internal
    - external (including library files)
  - Any instruction depend on piece of data
    - such as *load address instruction*

# Symbol Tables

- List of "items" in this asm/obj file that may be used by other asm/obj files
- What are they?
  - Labels: function calling
  - Data Labels: anything in `.data` section
    - Variables which may be accessed across files

# Producing an Object Module

Provides information for building a complete program from the pieces

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

- Header: described contents of object module
- Text segment: translated instructions
- Static data segment: data allocated for the life of the program
- Relocation info: for contents that depend on absolute location of loaded program
- Symbol table: global definitions and external refs
- Debug info: for associating with source code

# Contents

# Assembler & Linker

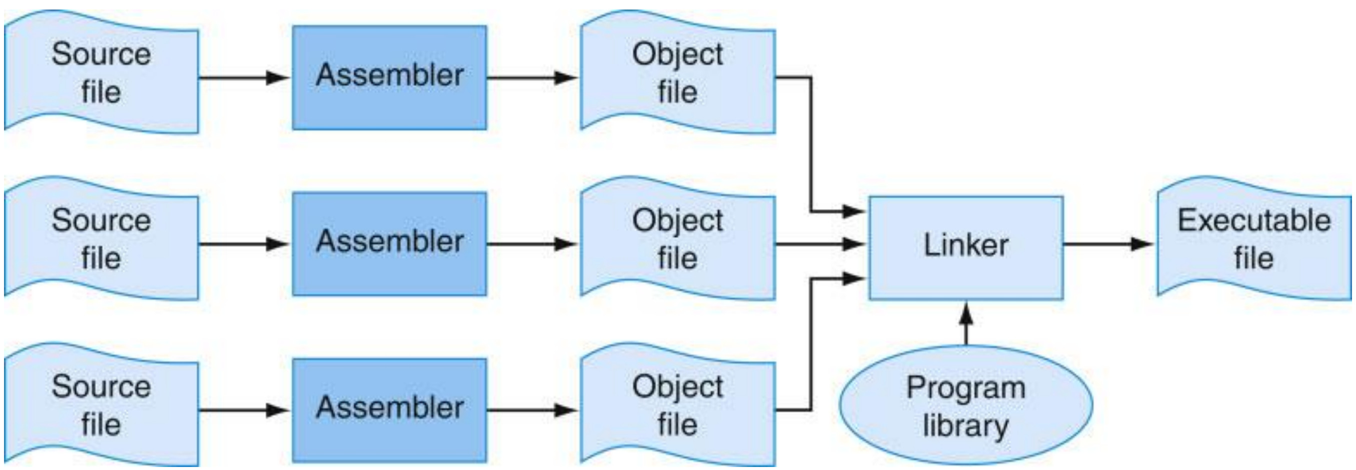
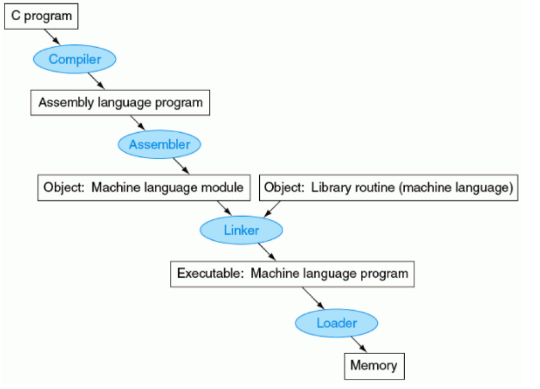



FIGURE A.1.1 **The process that produces an executable file.** An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

# Linker

- Combines object files together in order to produce an executable program
- Input
  - Object codes
  - Information tables
  - e.g. `foo.o,libc.o` for MIPS
- Output
  - Executable Code
    - e.g. `a.out` for MIPS



# Why need Linkers?

- Enable separate compilation of files
- Assume exe file directly generated by compiling and assembling a single code:
  - A single change to one line of a procedure
    - → compiling and assembling whole program
    - → Compiling library files each time

# Linker Primary Tasks

- Place all code and data modules together
- Search libraries to find library routines used by the program
- Determine memory locations for each module and relocate its instructions by adjusting absolute references
- Resolve any unresolved references among files including libraries

# Resolving References

- Search for reference (data or label) in all "user" symbol tables
- If not found, search library files
  - (for example, for `printf`)
- Once absolute address is determined, fill in machine code appropriately

# Linker Output

- Executable File similar to an Object File
  - containing text and data (plus header)
  - No unresolved references
  - No relocation information ?
    - Linker assumes first word of first text segment is at address `0x00000000`
  - No symbol tables
  - No debugging information

# Example: Static link of a C Program

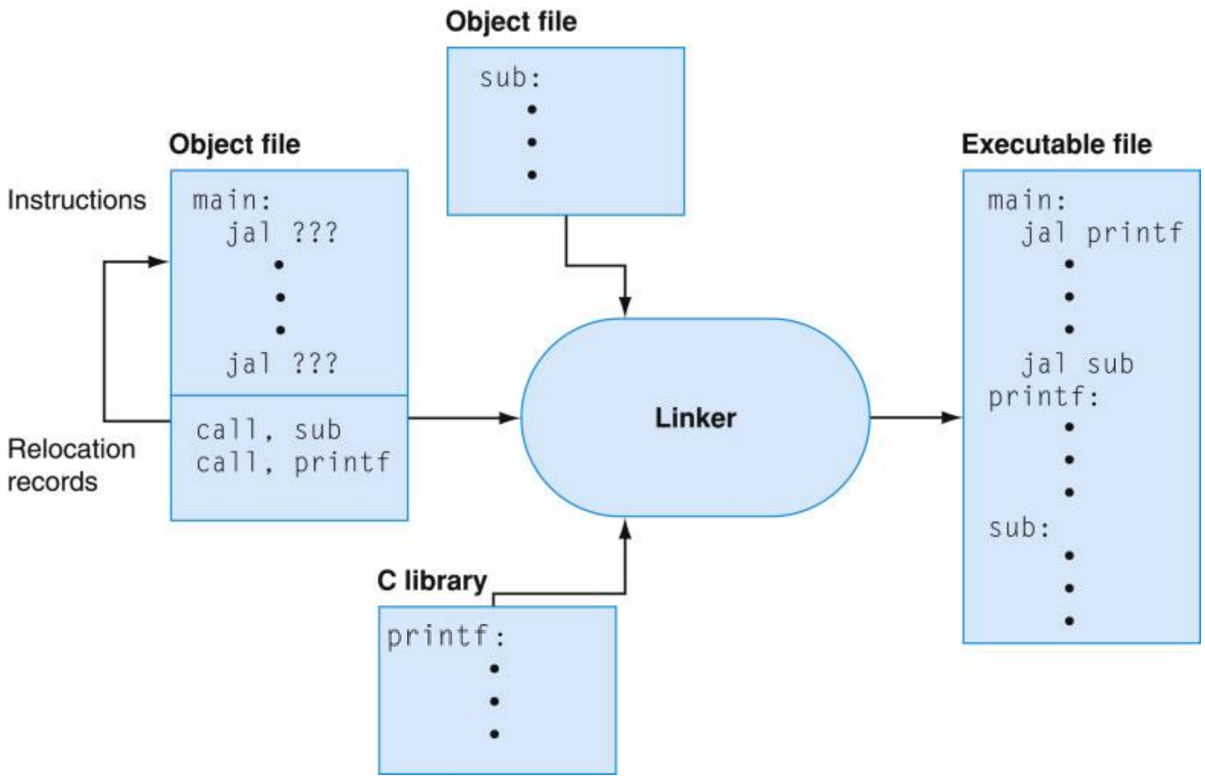



FIGURE A.3.1 **The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.**

# Static vs. Dynamic Link

- Statically Linked Library:
  - Libraries included by linker
- Dynamically Linked Library (DLL):
  - Library routines not linked (and loaded) until program is run

# Statically Linked Library

- Library routines part of exec. code
  - If a new version of library is released, it still keeps using old library version
    - New version to fix bugs or support new hardware devices
- Loads routines of library files used in exec. code all together
  - Even though those routines not executed
  - Library files can be very large

# Dynamically Linked Library (DLL)

- Pros
  - Storing a program requires less disk space
  - Sending a program requires less time
  - Executing two programs requires less memory (if they share a library)
  - Replacing one file (`libXYZ`) upgrades every program that use the library file
- Cons
  - Time overhead to do link at runtime
  - Unnecessary libraries are still linked (not in lazy DLL)

# DLL Linking

- Original DLL
  - Libraries linked once program is loaded
- Lazy DLL
  - Libraries linked during program execution and upon library call
- Advantages of Lazy DLL
  - Only links routines that are called during the running of the program not all library routines

# Summary of Linker Tasks

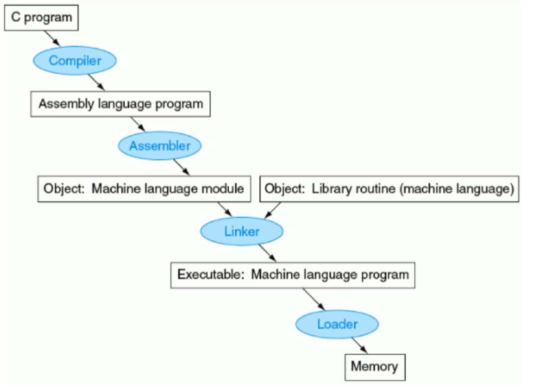
- Produce an executable image
  1. Merge segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external references
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

# Contents

- Introduction
- Compilers
- Assemblers
- Linkers
- **Loaders**
- Translation vs. Interpretation
- Java Applications

# Loader

- Loads an executable program into memory to be run
  - Executable files are stored on Disk
- Input:
  - Executable Code
    - (e.g., `a.out` for MIPS)
- Output:
  - (program is run)
- In reality, loader is the operating system

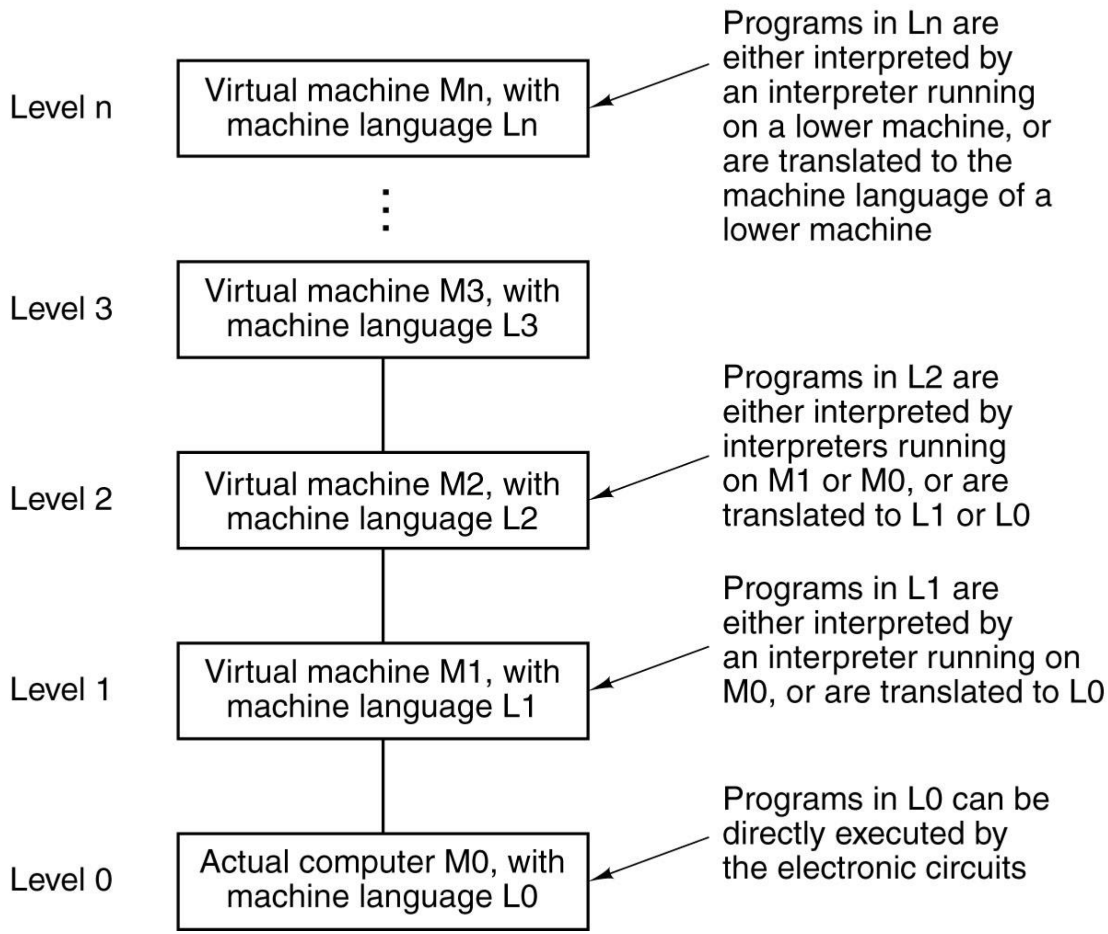


# Loading a Program

- Reads the executable file header
  - to determine size of the text and data segments
- Creates an address space large enough for the text and data
- Copies the instructions & data from the executable file into memory
- Copies the parameters (if any) to the main program onto the stack
- Initializes the machine registers (e.g. stack pointer)
- Jumps to a start-up routine
  - Copies the parameters into the argument registers
  - Calls the main routine of the program

# Contents

# Reminder



Level n
Virtual machine Mn, with machine language Ln
Programs in Ln are either interpreted by an interpreter running on a lower machine, or are translated to the machine language of a lower machine
Level 3
Virtual machine M3, with machine language L3
Level 2
Virtual machine M2, with machine language L2
Programs in L2 are either interpreted by interpreters running on M1 or M0, or are translated to L1 or L0
Level 1
Virtual machine M1, with machine language L1
Programs in L1 are either interpreted by an interpreter running on M0, or are translated to L0
Level 0
Actual computer M0, with machine language L0
Programs in L0 can be directly executed by the electronic circuits

# Translator vs. Interpreter

- Translator
  - Converts a program from source language to an equivalent program in another language
    - Like C compiler
- Interpreter
  - A program that executes other programs
  - A program that simulates an ISA
  - Directly executes a program in source language
    - Like MARS, Java Virtual Machine (JVM), Python interpreter

# Translation

- It is done offline
  - Before program execution
- Translated/compiled code almost always more efficient → higher performance
  - Performance important for many applications, particularly operating systems
- Compiled code can be only run on target machine (ISA dependent)
- Helps hiding program source from users

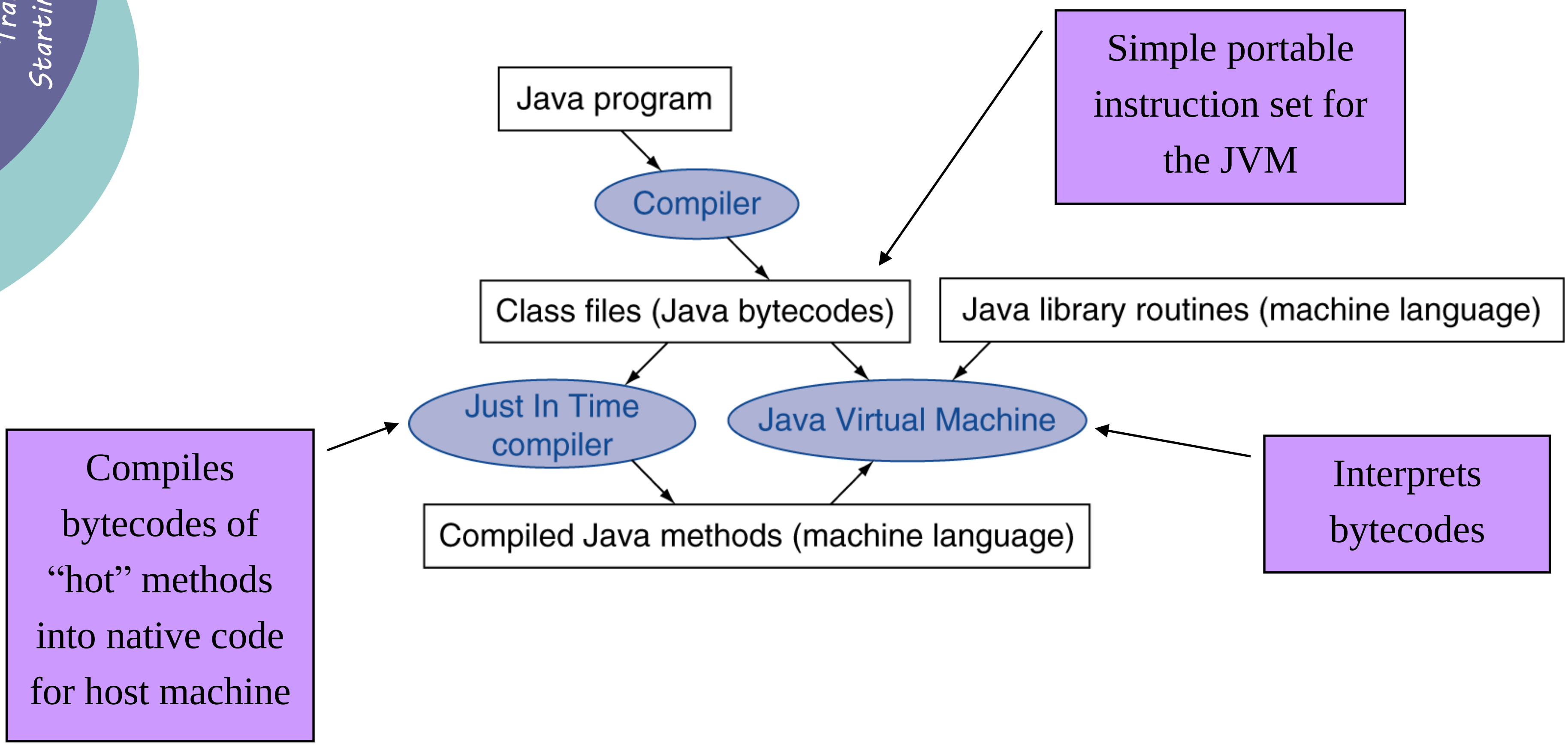
# Interpreting

- Performed online during program execution
- Used when performance not critical
- Typically 10x slower
- Smaller code size (2x)
- Provides instruction set independence
  - Can be run on any machine

# Contents

- Introduction
- Compilers
- Assemblers
- Linkers
- Loaders
- Translation vs. Interpretation
- Java Applications

# Starting Java Applications



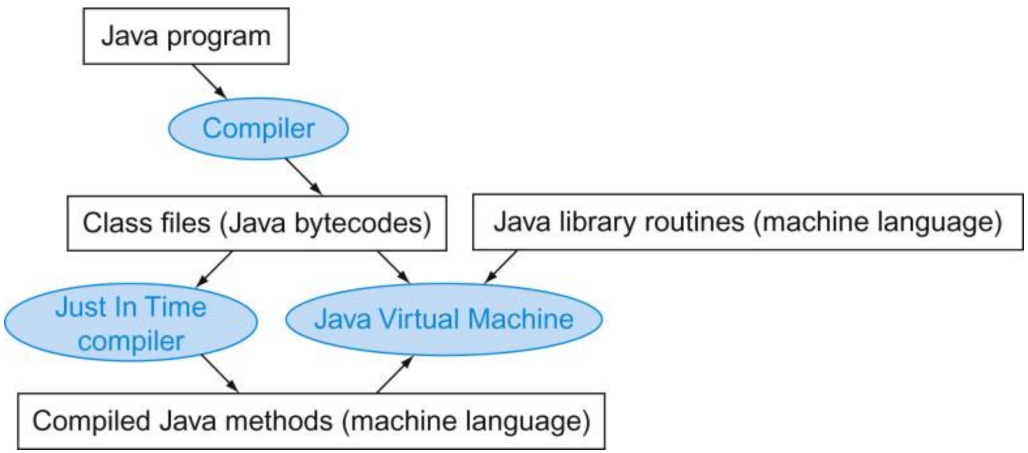
# Java Compilation

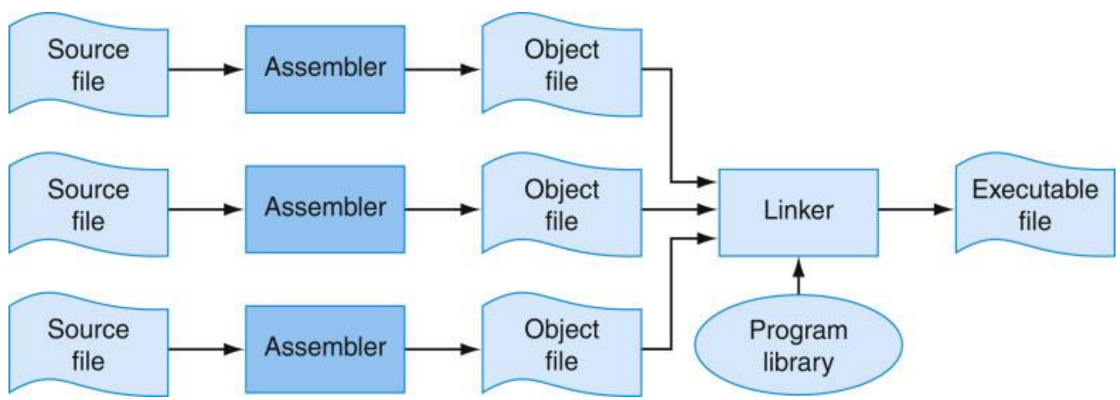
- Java Uses Interpreter
  - Java converted into "Java bytecodes"
  - Java bytecodes is executed on JVM
    - Java Virtual Machine
  - JVM translates Java bytecodes to machine language
- Advantage
  - Portability
- Disadvantage
  - Low performance

# Just-In-Time Compiler

- JIT or Just-In-Time Compiler
  - Operates at runtime
  - Translates interpreter code segments into machine language at runtime
  - Preserves portability & improves performance
    - Profile running program
    - Compiles hot methods
    - Save compiled portion for next run

# All-in-One


Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)


Source file
Assembler
Object file
Source file
Assembler
Object file
Source file
Assembler
Object file
Linker
Executable file
Program library