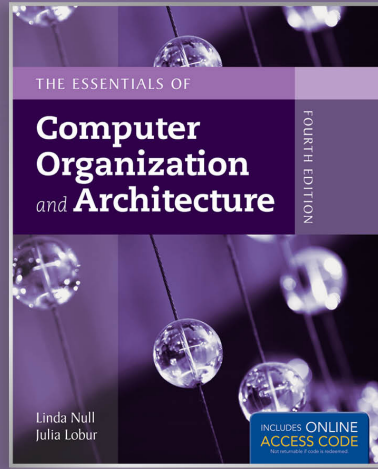


# Chapter 8

## System Software



© Odua Images/Shutterstock, Inc. Copyright © 2014 by Jones & Bartlett Learning, LLC an Ascend Learning Company  
www.jblearning.com

## Chapter 8 Objectives

- Become familiar with the functions provided by operating systems, programming tools, and database software.
- Understand the role played by each software component in maintaining the integrity of a computer system and its data.

2

## 8.1 Introduction

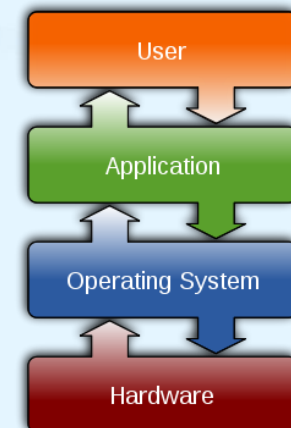
- The biggest and fastest computer in the world is of no use if it cannot efficiently provide beneficial services to its users.
- Users see the computer through their application programs. These programs are ultimately executed by computer hardware.
- System software -- **in the form of operating systems and middleware** -- is the glue that holds everything together.

3

## 8.2 Operating Systems

An **Operating System** is an **interface** between hardware and user which is responsible for the management and coordination of activities and the sharing of the resources of the computer.

The operating system must have control over the processor (as well as other resources).



4

## 8.2 Operating Systems

- The evolution of operating systems has paralleled the evolution of computer hardware.
  - As hardware became more powerful, operating systems allowed people to more easily manage the power of the machine.
- In the days when main memory was measured in kilobytes, and tape drives were the only form of magnetic storage, operating systems were simple *resident monitor* programs.
  - The resident monitor could only load, execute, and terminate programs.

5

## 8.2 Operating Systems

- In the 1960s, hardware has become powerful enough to accommodate *multiprogramming*, the concurrent execution of more than one task.
- Multiprogramming is achieved by allocating each process a given portion of CPU time (a *timeslice*).
- Interactive multiprogramming systems were called *timesharing* systems.
  - When a process is taken from the CPU and replaced by another, we say that a *context switch* has occurred.

6

## 8.2 Operating Systems

- Today, *multiprocessor systems* have become commonplace.
  - They present an array of challenges to the operating system designer, including the manner in which the processors will be synchronized, and how to keep their activities from interfering with each other.
- *Tightly coupled* multiprocessor systems share a common memory and the same set of I/O devices.
  - *Symmetric multiprocessor systems* (SMPs) are tightly coupled and load balanced (workload is distributed evenly).

7

## 8.2 Operating Systems

- *Loosely coupled* multiprocessor systems have physically separate memory.
  - These are often called *distributed systems*.
  - A type of distributed system is a networked system, which consists of a collection of interconnected, collaborating workstations.
- *Real time* operating systems control computers that respond to their environment (used for process control).
  - *Hard real time* systems have tight timing constraints, *soft real time* systems do not.

8

## 8.2 Operating Systems

- Personal computer operating systems are designed for ease of use rather than high performance.
- The idea that revolutionized small computer operating systems was the BIOS (Basic Input-output Operating System) chip that permitted a single operating system to function on different types of small systems.
  - The initial function of BIOS is to identify, test and initialize system devices such as keyboard, video display, hard disk and other hardware, and to load an operating system from a mass memory device. This process is known as **booting**.

9

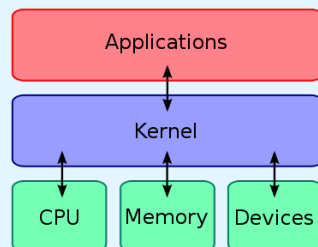
## 8.2 Operating Systems

- Operating systems having graphical user interfaces were first brought to market in the 1980s.
- At one time, these systems were considered appropriate only for desktop publishing and games. Today they are seen as technology enablers for users with little formal computer education.
- Once solely a server operating system, Linux holds the promise of bringing Unix to ordinary desktop systems (originally developed as a free operating system for x86-based personal computers).

10

## 8.2 Operating Systems

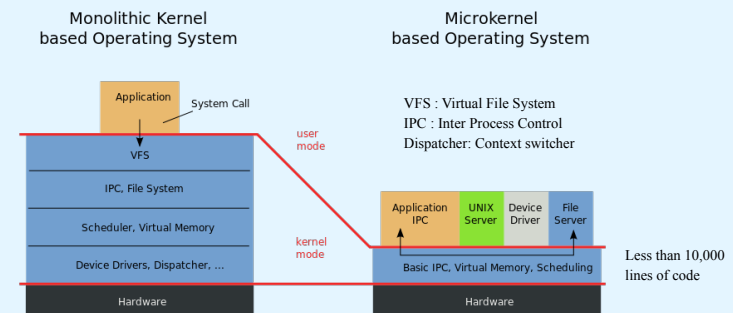
- Two operating system components are crucial: The **kernel** and the **system programs**.
- As the core of the operating system, the kernel performs **scheduling, synchronization, memory management, interrupt handling**, and it provides **protection**.



11

## 8.2 Operating Systems

- **Monolithic** systems provide most of their services within a single operating system program
- **Microkernel** systems provide minimal functionality, with most services carried out by external programs



12

## 8.2 Operating Systems

- **Microkernel** systems provide better security, easier maintenance, and portability at the expense of execution speed.
  - Examples are MINIX, Mach, and QNX.
  - Symmetric multiprocessor computers are ideal platforms for microkernel operating systems.
- **Monolithic** systems give faster execution speed, but are difficult to port from one architecture to another.
  - Examples are Linux, MacOS, and DOS.

13

## 8.2 Operating Systems

- Process management lies at the heart of operating system services.
  - The operating system creates processes, schedules their access to resources, deletes processes, and deallocates resources that were allocated during process execution.
- The operating system monitors the activities of each process to avoid **synchronization** problems that can occur when processes use shared resources.
- If processes need to **communicate** with one another, the operating system provides the services.

14

## 8.2 Operating Systems

- The operating system schedules process execution.
- First, the operating system determines which processes shall be granted access to the CPU.
  - This is **long-term scheduling**.
- After a number of processes have been admitted, the operating system determines which one will have access to the CPU at any particular moment.
  - This is **short-term scheduling**.
- **Context switches** occur when a process is taken from the CPU and replaced by another process.
  - Information relating to the state of the process is preserved during a context switch. For instance, CPU registers and page tables must be saved.

15

## 8.2 Operating Systems

- Short-term scheduling can be **nonpreemptive** or **preemptive**.
- In **nonpreemptive** scheduling, a process has use of the CPU until either it terminates, or must wait for resources that are temporarily unavailable.
- In **preemptive** scheduling, each process is allocated a timeslice. When the timeslice expires, a context switch occurs.
- A context switch can also occur when a higher-priority process needs the CPU.

16

## 8.2 Operating Systems

- Four approaches to CPU scheduling are:
  - **First-come, first-served** where jobs are serviced in arrival sequence and run to completion if they have all of the resources they need.
  - **Shortest job first** where the smallest jobs get scheduled first. (The trouble is in knowing which jobs are shortest!)
  - **Round robin** scheduling where each job is allotted a certain amount of CPU time. A context switch occurs when the time expires.
  - **Priority** scheduling preempts a job with a lower priority when a higher-priority job needs the CPU.

17

## 8.2 Operating Systems

SLOC: Source Lines Of Code

Year	Operating System	SLOC (Million)
1993	Windows NT 3.1	4-5 <sup>[1]</sup>
1994	Windows NT 3.5	7-8 <sup>[1]</sup>
1996	Windows NT 4.0	11-12 <sup>[1]</sup>
2000	Windows 2000	more than 29 <sup>[1]</sup>
2001	Windows XP	45 <sup>[2],[3]</sup>
2003	Windows Server 2003	50 <sup>[1]</sup>

Operating System	SLOC (Million)
Debian 2.2	55-59 <sup>[4],[5]</sup>
Debian 3.0	104 <sup>[5]</sup>
Debian 3.1	215 <sup>[5]</sup>
Debian 4.0	283 <sup>[5]</sup>
Debian 5.0	324 <sup>[5]</sup>
OpenSolaris	9.7
FreeBSD	8.8
Mac OS X 10.4	86 <sup>[6]</sup>
Linux kernel 2.6.0	5.2
Linux kernel 2.6.29	11.0
Linux kernel 2.6.32	12.6 <sup>[7]</sup>
Linux kernel 2.6.35	13.5 <sup>[8]</sup>
Linux kernel 3.6	15.9 <sup>[9]</sup>

18

## 8.3 Protected Environments

- In their role as resource managers and protectors, many operating systems provide protected environments that isolate processes, or groups of processes from each other.
- Three common approaches to establishing protected environments are **virtual machines**, **subsystems**, and **partitions**.
- These environments simplify system management and control, and can provide emulated machines to enable execution of programs that the system would otherwise be unable to run.

19

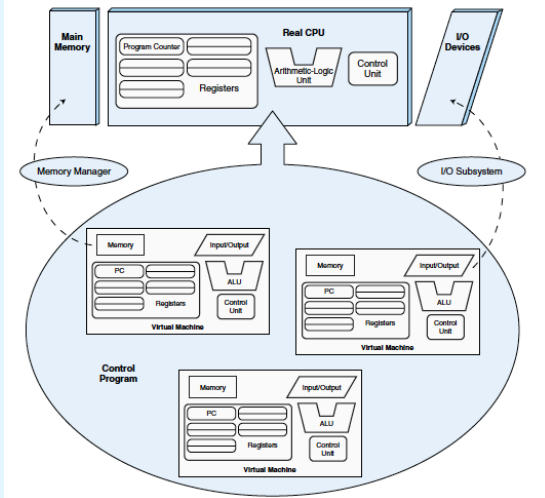
## 8.3 Protected Environments

- **Virtual machines** are a protected environment that presents an image of itself -- or the image of a totally different architecture -- to the processes that run within the environment.
- A virtual machine is exactly that: **an imaginary computer**.
- The underlying real machine is under the control of the kernel. The kernel receives and manages all resource requests that emit from processes running in the virtual environment.

The next slide provides an illustration.

20

## 8.3 Protected Environments



21

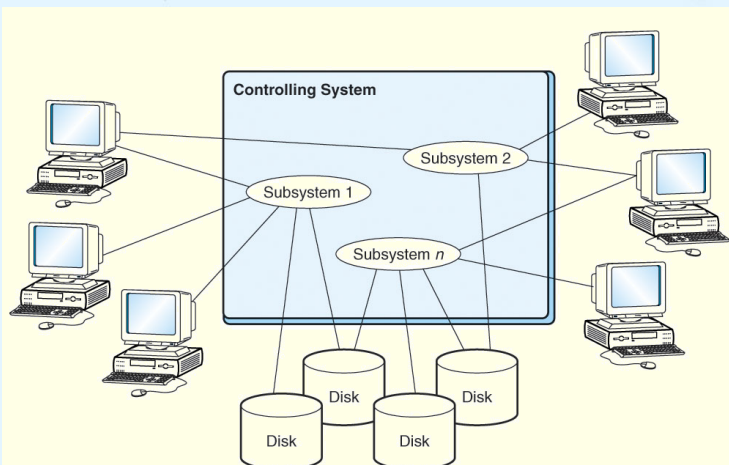
## 8.3 Protected Environments

- **Subsystems** are another type of protected environment.
- They provide logically distinct environments that can be individually controlled and managed. They can be stopped and started independent on each other.
  - Subsystems can have special purposes, such as controlling I/O or virtual machines. Others partition large application systems to make them more manageable.
  - In many cases, resources must be made visible to the subsystem before they can be accessed by the processes running within it.

The next slide provides an illustration.

22

## 8.3 Protected Environments



23

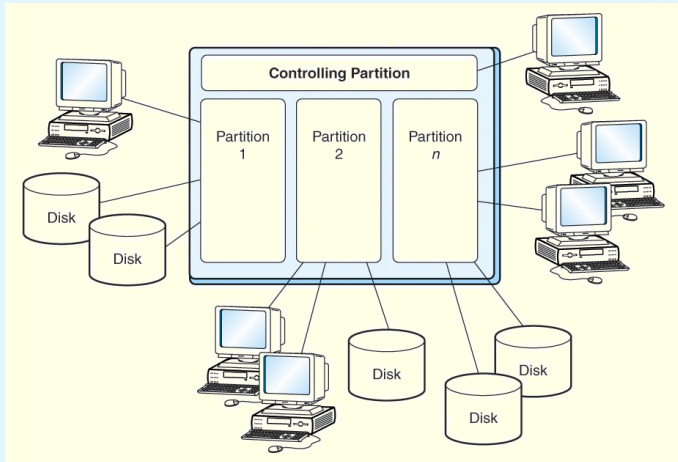
## 8.3 Protected Environments

- In very large computers, subsystems do not go far enough to establish a protected environment.
- **Logical partitions (LPARs)** provide much higher barriers: Processes running within a logical partition have no access to processes running in another partition unless a connection between them (e.g., FTP) is explicitly established.
- LPARs are an enabling technology for the recent trend of consolidating hundreds of small servers within the confines of a single large system.

The next slide provides an illustration.

24

## 8.3 Protected Environments



25

## 8.4 Programming Tools

- **Programming tools** carry out the mechanics of software creation within the confines of the operating system and hardware environment.
- **Assemblers** are the simplest of all programming tools. They translate mnemonic instructions to machine code.
- Most assemblers carry out this translation in **two passes** over the source code.
  - The first pass partially assembles the code and builds the symbol table
  - The second pass completes the instructions by supplying values stored in the symbol table.

26

## 8.4 Programming Tools

- Example MARIE Code

Assembled version is on top right. The “+” indicates relative offset.

```

Load x           1+004
Add y            3+005
Store z         2+006
Halt            7000
x, DEC 35       0023
y, DEC -23     FFE9
z, HEX 0000    0000
    
```

Address	Memory Contents
0x250	1254
0x251	3255
0x252	2256
0x253	7000
0x254	0023
0x255	FFE9
0x256	0000

Loaded Starting at Address 0x250

Address	Memory Contents
0x400	1404
0x401	3405
0x402	2406
0x403	7000
0x404	0023
0x405	FFE9
0x406	0000

Loaded Starting at Address 0x400

27

## 8.4 Programming Tools

- The output of most assemblers is a stream of **relocatable** binary code.
  - In relocatable code, operand addresses are relative to where the operating system chooses to load the program.
  - Absolute (nonrelocatable) code is most suitable for device and operating system control programming.
- When relocatable code is loaded for execution, special registers provide the base addressing.
- Addresses specified within the program are interpreted as offsets from the base address.

28

## 8.4 Programming Tools

- The process of assigning physical addresses to program variables is called *binding*.
- Binding can occur at compile time, load time, or run time.
- **Compile time binding** gives us absolute code.
- **Load time binding** assigns physical addresses as the program is loaded into memory.
  - With load time, binding the program cannot be moved!
- **Run time binding** requires a base register to carry out the address mapping.

29

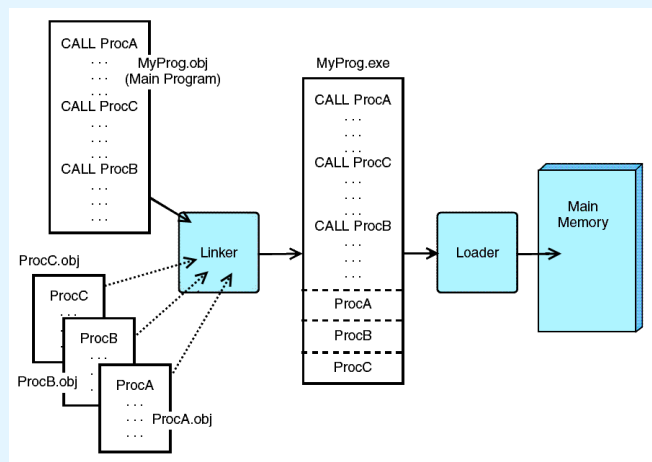
## 8.4 Programming Tools

- On most systems, binary instructions must pass through a *link editor* (or *linker*) to create an executable module.
- Link editors incorporate various binary routines into a single executable file as called for by a program's external symbols.
- Like assemblers, link editors perform two passes: The first pass creates a symbol table and the second resolves references to the values in the symbol table.

The next slide shows this process schematically.

30

## 8.4 Programming Tools



31

## 8.4 Programming Tools

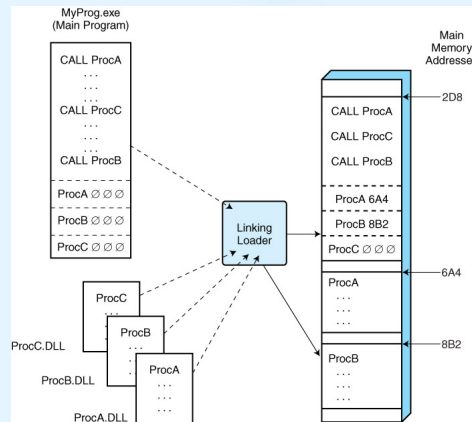
- **Dynamic linking** is when the link editing is delayed until load time or at run time.
- External modules are loaded from *dynamic link libraries* (DLLs).
- **Load time dynamic linking** slows down program loading, but calls to the DLLs are faster.
- **Run time dynamic linking** occurs when an external module is first called, causing slower execution time.
- Dynamic linking makes program modules smaller, but carries the risk that the programmer may not have control over the DLL.

32



## 8.4 Programming Tools

- Dynamic Linking with Load Time Address Resolution



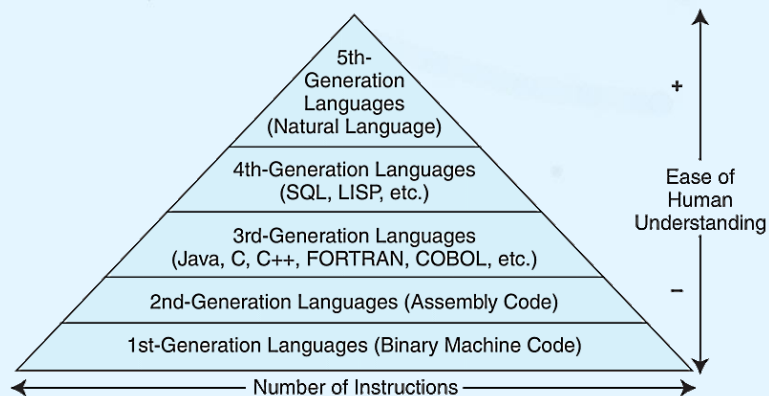
33

## 8.4 Programming Tools

- Assembly language is considered a “**second generation**” programming language (2GL).
- Compiled programming languages, such as C, C++, Pascal, and COBOL, are “**third generation**” languages (3GLs).
- Each language generation presents problem solving tools that are closer to how people think and farther away from how the machine implements the solution.

34

## 8.4 Programming Tools



Keep in mind that the computer can understand only the 1GL!

35

## 8.4 Programming Tools

- **Compilers** bridge the semantic gap between the higher level language and the machine's binary instructions.
- Most compilers effect this translation in a six-phase process. The first three are **analysis** phases:
  1. **Lexical analysis** extracts tokens, e.g., reserved words and variables.
  2. **Syntax analysis** (parsing) checks statement construction.
  3. **Semantic analysis** checks data types and the validity of operators.

36

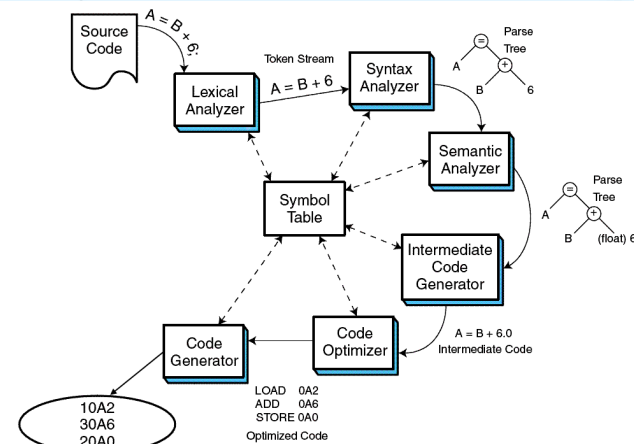
## 8.4 Programming Tools

- The last three compiler phases are **synthesis** phases:
  - Intermediate code generation** creates *three address code* to facilitate optimization and translation.
  - Optimization** creates assembly code while taking into account architectural features that can make the code efficient.
  - Code generation** creates binary code from the optimized assembly code.
- Through this modularity, compilers can be written for various platforms by rewriting only the last two phases.

The next slide shows this process graphically.

37

## 8.4 Programming Tools



38

## 8.4 Programming Tools

- Interpreters** produce executable code from source code in real time, one line at a time.
- Consequently, this not only makes interpreted languages slower than compiled languages but it also affords less opportunity for error checking.
- Interpreted languages are, however, very useful for teaching programming concepts, because feedback is nearly instantaneous, and performance is rarely a concern.

39

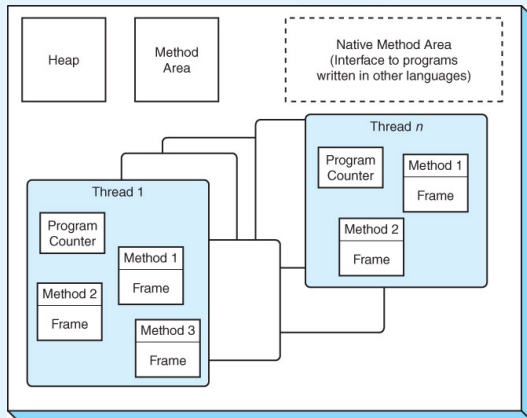
## 8.5 Java: All of the Above

- The Java programming language exemplifies many of the concepts that we have discussed in this chapter.
- Java programs (*classes*) execute within a virtual machine, the **Java Virtual Machine** (JVM).
- This allows the language to run on any platform for which a virtual machine environment has been written.
- Java is both a compiled and an interpreted language. The output of the compilation process is an assembly-like intermediate code (**bytecode**) that is interpreted by the JVM.

40

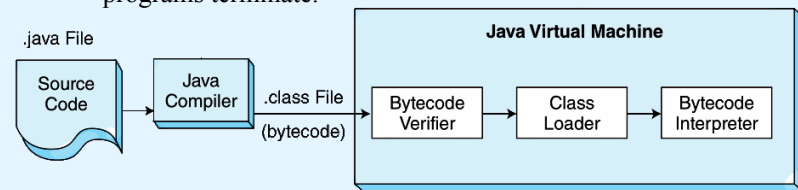
## 8.5 Java: All of the Above

- The Java Virtual Machine



## 8.5 Java: All of the Above

- The JVM is an operating system in miniature.
  - It loads programs, links them, starts execution threads, manages program resources, and deallocates resources when the programs terminate.



- Because the JVM performs so many tasks at run time, its performance cannot match the performance of a traditional compiled language.

## 8.5 Java: All of the Above

- At execution time, a Java Virtual Machine must be running on the host system.
- It loads and executes the bytecode class file.
- While loading the class file, the JVM verifies the integrity of the bytecode.
- The loader then performs a number of run-time checks as it places the bytecode in memory.
- The loader invokes the bytecode interpreter.

43

## 8.5 Java: All of the Above

### The bytecode interpreter:

1. Performs a link edit of the bytecode instructions by asking the loader to supply all referenced classes and system binaries, if they are not already loaded.
2. Creates and initializes the main stack frame and local variables.
3. Creates and starts execution thread(s).
4. Manages heap storage by deallocating unused storage while the threads are executing.
5. Deallocates resources of terminated threads.
6. Upon program termination, kills any remaining threads and terminates the JVM.

44

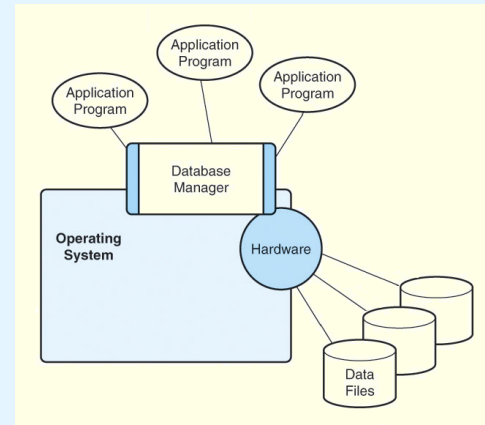
## 8.5 Java: All of the Above

- Because the JVM does so much as it loads and executes its bytecode, it can't match the performance of a compiled language.
  - This is true even when speedup software like Java's Just-In-Time (JIT) compiler is used.
- However class files can be created and stored on one platform and executed on a completely different platform.
- This "write once, run-anywhere" paradigm is of enormous benefit for enterprises with disparate and geographically separate systems.
- Given its portability and relative ease of use, the Java language and its virtual machine environment are the ideal middleware platform.

45

## 8.6 Database Software

- Database systems contain the most valuable assets of an enterprise. They are the foundation upon which application systems are built.



A **database** is an organized collection of data.

46

## 8.6 Database Software

- Database systems provide a single definition, the **database schema**, for the data elements that are accessed by application programs.
  - A **logical schema** is the application program's view of the database that defines field sizes and data types.
  - A **physical schema** is the computer's view of the database that includes locations of physical files and indexes.
- Within the logical schema, certain data fields are designated as record **keys** that provide efficient access to records in the database.

47

## 8.6 Database Software

- Keys are stored in physical **index file structures** containing pointers to the location of the physical records.
- Many implementations use a variant of a B+ tree for index management because B+ trees can be optimized with consideration to the I/O system and the applications.
- In many cases, the "higher" nodes of the tree will persist in cache memory, requiring physical disk accesses only when traversing the lower levels of the index.

48

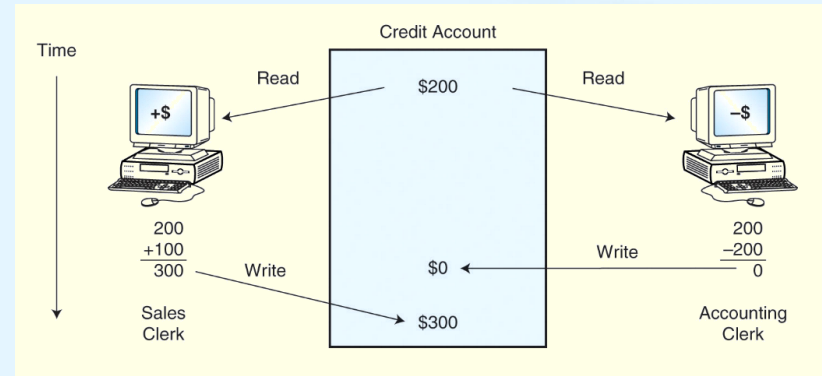
## 8.6 Database Software

- Most database systems also include transaction management components to assure that the database is always in a consistent state.
- Transaction management provides the following properties:
  - **Atomicity** - All related updates occur or no updates occur.
  - **Consistency** - All updates conform to defined data constraints.
  - **Isolation** - No transaction can interfere with another transaction.
  - **Durability** - Successful updates are written to durable media as soon as possible.
- These are the **ACID properties** of transaction management.

49

## 8.6 Database Software

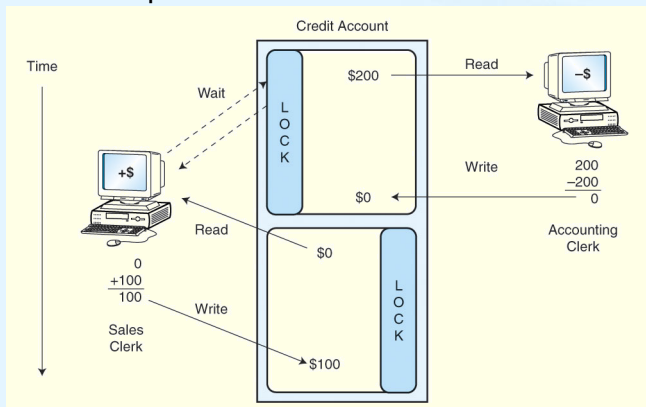
- Without the ACID properties, **race conditions** can occur:



50

## 8.6 Database Software

- Record locking mechanisms assure **isolated, atomic** database updates:



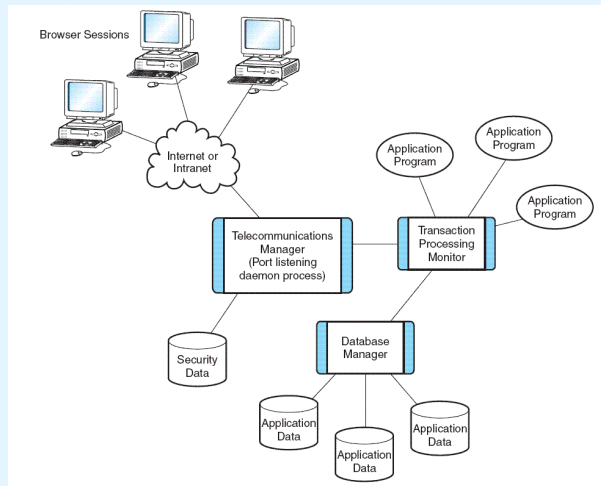
51

## 8.7 Transaction Managers

- One way to improve database performance is to ask it to do less work by moving some of its functions to specialized software.
- Transaction management is one component that is often partitioned from the core database system.
- Transaction managers are especially important when the transactions involve more than one physical database, or the application system spans more than one class of computer, as in a multitiered architecture.
- One of the most widely-used transaction management systems is CICS (Customer Information Control System) shown on the next slide.

52

## 8.7 Transaction Managers



53

## Chapter 8 Conclusion

- The proper functioning and performance of a computer system depends as much on its software as its hardware.
- The operating system is the system software component upon which all other software rests.
- Operating systems control process execution, resource management, protection, and security.
- Subsystems and partitions provide compatibility and ease of management.

54

## Chapter 8 Conclusion

- Programming languages are often classed into generations, with assembly language being the first generation.
- All languages above the machine level must be translated into machine code.
- Compilers bridge this semantic gap through a series of six steps.
- Link editors resolve system calls and external routines, creating a unified executable module.

55

## Chapter 8 Conclusion

- The Java programming language incorporates the idea of a virtual machine, a compiler and an interpreter.
- Database software provides controlled access to data files through enforcement of ACID properties.
- Transaction managers provide high performance and cross-platform access to data.

56

End of Chapter 8