Chapter 6: An Introduction to System Software and Virtual Machines

Invitation to Computer Science,
Objectives

In this chapter, you will learn about:

- System software
- Assemblers and assembly language
- Operating systems
Introduction

- Von Neumann computer
  - “Naked machine”
  - Hardware without any helpful user-oriented features
  - Extremely difficult for a human to work with

- An interface between the user and the hardware is needed to make a Von Neumann computer usable
Introduction (continued)

- Tasks of the interface
  - Hide details of the underlying hardware from the user
  - Present information in a way that does not require in-depth knowledge of the internal structure of the system
Introduction (continued)

- Tasks of the interface (continued)
  - Allow easy user access to the available resources
  - Prevent accidental or intentional damage to hardware, programs, and data
System Software: The Virtual Machine

- System software
  - Acts as an intermediary between users and hardware
  - Creates a virtual environment for the user that hides the actual computer architecture

- Virtual machine (or virtual environment)
  - Set of services and resources created by the system software and seen by the user
Figure 6.1
The Role of System Software
Types of System Software

- System software is a collection of many different programs
- Operating system
  - Controls the overall operation of the computer
  - Communicates with the user
  - Determines what the user wants
  - Activates system programs, applications packages, or user programs to carry out user requests
Figure 6.2
Types of System Software
Types of System Software (continued)

- User interface
  - Graphical user interface (GUI) provides graphical control of the capabilities and services of the computer

- Language services
  - Assemblers, compilers, and interpreters
  - Allow you to write programs in a high-level, user-oriented language, and then execute them
Types of System Software (continued)

- Memory managers
  - Allocate and retrieve memory space

- Information managers
  - Handle the organization, storage, and retrieval of information on mass storage devices

- I/O systems
  - Allow the use of different types of input and output devices
Types of System Software (continued)

- **Scheduler**
  - Keeps a list of programs ready to run and selects the one that will execute next

- **Utilities**
  - Collections of library routines that provide services either to user or other system routines
Instructions And Programs

- Each computer model has its own machine language.
  - The machine instruction format is designed by the computer designer
  - The format chosen for an instruction determines the number of operations
    - directly supported in hardware (called hardwired instructions) and
    - the size of the addressing space.
Machine Language Programming

- Machine language
  - Uses binary
  - Allows only numeric memory addresses
  - Difficult to create data
  - Difficult to change
Machine Language (continued)

- Difficult to change:

Suppose you have written this program (for clarity I use mnemonics instead of opcodes)

0: load 4
1: add 5
2: store 6
3: halt
4: .data 5
5: .data 3
6: .data 10

Now you want to modify the program and add an increment to what was just stored

The modified program

0: load 5
1: add 6
2: store 7
3: increase 7
4: halt
5: .data 5
6: .data 3
7: .data 10

Note! I had to rewrite almost all the addresses!!!
Computers can only execute machine language programs!

Although humans CAN program in a machine language, the difficulties of using the machine language makes them hard to use for writing programs.

Remember:

• Writing and reading binary numbers is error prone and difficult.
  • (Hexadecimal notation helps, but it doesn't eliminate the problems).
• Converting data and addresses to binary form is not fun.
Assemblers and Assembly Language: Assembly Language

- Assembly languages

  - Designed to overcome shortcomings of machine languages
  - Create a more productive, user-oriented environment
  - Earlier termed second-generation languages
  - Now viewed as low-level programming languages
Figure 6.3
The Continuum of Programming Languages
Assembly Language (continued)

- Source program
  - An assembly language program

- Object program
  - A machine language program

- Assembler
  - Translates a source program into a corresponding object program
Figure 6.4
The Translation/Loading/Execution Process
Translation and Loading

- Before a source program can be run, an assembler and a loader must be invoked

Assembler

- Translates a symbolic assembly language program into machine language

Loader

- Reads instructions from the object file and stores them into memory for execution
Translation and Loading (continued)

Assembler tasks

- Convert symbolic op codes to binary
- Convert symbolic addresses to binary
- Perform assembler services requested by the pseudo-ops
- Put translated instructions into a file for future use
Translation: Assembler

Input:

```assembly
.bEGIN
    load x
    add y
    store z
    increment z
    halt
x:  .DATA 5
y:  .DATA 3
z:  .DATA 10
.end
```

Output: the symbol table

<table>
<thead>
<tr>
<th>label</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
</tr>
<tr>
<td>z</td>
<td>7</td>
</tr>
</tbody>
</table>

Our program

0: load 5
1: add 6
2: store 7
3: increase 7
4: halt
5: data 5
6: data 3
7: data 10
Assembler

Source Code

```
.begin
  load x
  add y
  store z
  increment z
  halt
.x: .data 5
.y: .data 3
.z: .data 10
.end
```

Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
</tr>
<tr>
<td>z</td>
<td>7</td>
</tr>
</tbody>
</table>

Object Code

<table>
<thead>
<tr>
<th>Address</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0011</td>
<td>0000</td>
<td>0000</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>0001</td>
<td>0000</td>
<td>0000</td>
<td>0111</td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td>0000</td>
<td>0000</td>
<td>0111</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0101</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>1100</td>
</tr>
</tbody>
</table>

Label location

- x: 5
- y: 6
- z: 7
Assembly Language (continued)

- Advantages of writing in assembly language rather than machine language
  1. Use of symbolic operation codes rather than numeric (binary) ones
  2. Use of symbolic memory addresses rather than numeric (binary) ones
  3. Pseudo-operations that provide useful user-oriented services such as data generation
Figure 6.6
Structure of a Typical Assembly Language Program
Examples of Assembly Language Code (continued)

- Algorithmic operations

  Set the value of $i$ to 1 (line 2).

  : 

  Add 1 to the value of $i$ (line 7).
Examples of Assembly Language Code (continued)

- Assembly language translation

```
LOAD    ONE    --Put a 1 into register R.
STORE   I      --Store the constant 1 into i.

INCREMENT I    --Add 1 to memory location i.

I:    .DATA    0    --The index value. Initially it is 0.
ONE:   .DATA    1    --The constant 1.
```
Examples of Assembly Language Code (continued)

- Arithmetic expression

\[ A = B + C - 7 \]

(Assume that B and C have already been assigned values)

- This correspond to the pseudo code instruction:
  
  *Set A to B plus C minus 7*
Examples of Assembly Language Code (continued)

- **Assembly language translation**

```
.BEGIN
  LOAD B   --Put the value B into register R.
  ADD C    --R now holds the sum (B + C).
  SUBTRACT SEVEN --R now holds the expression (B + C - 7).
  STORE A   --Store the result into A.
  HALT     --The data are placed after the HALT

A: .DATA 0
B: .DATA 0
C: .DATA 0
SEVEN: .DATA 7 --The constant 7.
.END
```
Examples of Assembly Language Code (continued)

- Arithmetic Expression Program - **MODIFICATION 1**

- Now I want to ask the user to provide for me the values of B, and C. Then I want to output the value of A.
  - This corresponds to the pseudo code:
    - Get the value of B and C
    - Print the value of A

- IN and OUT are the mnemonics for getting data from input or producing a data in output
  - Note that in our simulated computer numbers are converted in characters in order to be printed since the screen displays ASCII codes!
Examples of Assembly Language Code (continued)

- Assembly language translation

  .BEGIN

  IN B        --Get the value of B from the keyboard and put it in B
  IN  C       --Get the value of C from the keyboard and put it in C
  LOAD B      --Put the value B into register R.
  ADD C       --R now holds the sum (B + C).
  SUBTRACT SEVEN   --R now holds the expression (B + C - 7).
  STORE A     --Store the result into A.
  OUT A       --Output the value of A
  HALT

  --The data are placed after the HALT

  A:  .DATA  0
  B:  .DATA  0
  C:  .DATA  0
  SEVEN: .DATA 7  --The constant 7.

  .END
Examples of Assembly Language Code (continued)

- Arithmetic Expression Program - **MODIFICATION 2**

- The second change consists in checking if \( B \) is equal to \( C \). If \( B = C \), I want to add 7 to \( B \), otherwise, I want to add 1 to \( C \). Then set \( A = B + C - 7 \)

  - This corresponds to the pseudo code:

    ```
    If (B = C) then
      Set B to B+7
    otherwise
      Set C to C+1
    Set A to B+C-7
    ```
If-Then-Else in the program

.BEGIN
    IN B  --Get the value of B and put it in B
    IN  C  --Get the value of C and put it in C
    LOAD  B
    COMPARE C  -- Tests if B=C
    JUMPNEQ ELSE  -- If B≠C jump to the label ELSE
    ADD SEVEN  -- B=C so add 7 to B
    STORE B
    JUMP OUTIF  -- go out of the if instruction
ELSE:
    INCREMENT C
OUTIF:
    LOAD  B  -- (optional since B is already in R)
    ADD  C  --R now holds the sum (B + C).
    SUBTRACT SEVEN  --R now holds the expression (B + C - 7).
    STORE A  --Store the result into A.
    OUT  A  --Output the value of A
    HALT  --The data are placed after the HALT
A: .DATA 0
B: .DATA 0
C: .DATA 0
SEVEN: .DATA 7  --The constant 7.
.END
Examples of Assembly Language Code (continued)

- Arithmetic Expression Program - MODIFICATION 3

- Remove the last changes and replace them with a while loop. While B<C, I want to subtract 7 from C, then increment B. After the while is terminated, set A=B+C-7.

  - This corresponds to the pseudo code:

```
while (B < C)
    Set C to C-7
    Set B=B+1
endwhile
Set A to B+C-7
```
While loop in the program

BEGIN

IN B
IN C

LOOP: LOAD B
COMPARE C
JUMPLT ENDLOOP
JUMPEQ ENDLOOP
LOAD C
SUBTRACT SEVEN
STORE C
INCREMENT B
JUMP LOOP

ENDLOOP: LOAD B
ADD C
SUBTRACT SEVEN
STORE A
OUT A
HALT

A: .DATA 0
B: .DATA 0
C: .DATA 0
SEVEN: .DATA 7 --The constant 7.

.END
This Exercise is for you to complete!

- Arithmetic Expression Program - MODIFICATION 4

Instead of the while loop use a repeat-until loop. Repeat “subtract 7 from C, then increment B, until B ≥ C. When the repeat is terminated, set A=B+C-7.

This corresponds to the pseudo code:

```
repeat
  Set C to C-7
  Set B=B+1
until (B ≥ C)
Set A to B+C-7
```
Examples of Assembly Language Code (continued)

- Problem
  - Read in a sequence of non-negative numbers, one number at a time, and compute a running sum
  - When you encounter a negative number, print out the sum of the non-negative values and stop
<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Set the value of Sum to 0</td>
</tr>
<tr>
<td>2</td>
<td>Input the first number $N$</td>
</tr>
<tr>
<td>3</td>
<td>While $N$ is not negative do</td>
</tr>
<tr>
<td>4</td>
<td>Add the value of $N$ to Sum</td>
</tr>
<tr>
<td>5</td>
<td>Input the next data value $N$</td>
</tr>
<tr>
<td>6</td>
<td>End of the loop</td>
</tr>
<tr>
<td>7</td>
<td>Print out Sum</td>
</tr>
<tr>
<td>8</td>
<td>Stop</td>
</tr>
</tbody>
</table>

Figure 6.7
Algorithm to Compute the Sum of Numbers
Fig.6.8 - Assembly Language Program to Compute the Sum of Nonnegative Numbers

```
.BEGIN
CLEAR     SUM   --This marks the start of the program.
IN        N     --Set the running sum to 0 (line 1).

--The next three instructions test whether N is a negative number (line 3).
AGAIN:   LOAD    ZERO  --Put 0 into register R.
         COMPARE  N     --Compare N and 0.
         JUMPLT   NEG    --Go to NEG if N < 0.

--We get here if N ≥ 0. We add N to the running sum (line 4).
LOAD     SUM     --Put SUM into R.
ADD      N       --Add N. R now holds (N + SUM).
STORE    SUM     --Put the result back into SUM.

--Get the next input value (line 5).
IN        N

--Now go back and repeat the loop (line 6).
JUMP     AGAIN

--We get to this section of the program only when we encounter a negative value.
NEG:      OUT     SUM   --Print the sum (line 7)
          HALT    --and stop (line 8).

--Here are the data generation pseudo-ops
SUM:      .DATA    0     --The running sum goes here.
N:        .DATA    0     --The input data are placed here.
ZERO:     .DATA    0     --The constant 0.

--Now we mark the end of the entire program.
.END
```
Operating Systems

- System commands
  - Carry out services such as translate a program, load a program, run a program
  - Types of system commands
    - Lines of text typed at a terminal
    - Menu items displayed on a screen and selected with a mouse and a button: point-and-click
  - Examined by the operating system
Functions of an Operating System

- Five most important responsibilities of the operating system
  - User interface management
  - Program scheduling and activation
  - Control of access to system and files
  - Efficient resource allocation
  - Deadlock detection and error detection
The User Interface

- Operating system
  - Waits for a user command
  - If command is legal, activates and schedules the appropriate software package

- User interfaces
  - Text-oriented
  - Graphical
Figure 6.15
User Interface
Responsibility of the Operating System
System Security And Protection

- The operating system must prevent
  - Non-authorized people from using the computer
    - User names and passwords
  - Legitimate users from accessing data or programs they are not authorized to access
    - Authorization lists
Efficient Allocation Of Resources

- The operating system ensures that
  - Multiple tasks of the computer may be underway at one time
  - Processor is constantly busy
    - Keeps a “queue” of programs that are ready to run
    - Whenever processor is idle, picks a job from the queue and assigns it to the processor
The Safe Use Of Resources

- **Deadlock**
  - Two processes are each holding a resource the other needs
  - Neither process will ever progress

- The operating system must handle deadlocks
  - Deadlock prevention
  - Deadlock recovery
Historical Overview of Operating Systems Development

- First generation of system software (roughly 1945–1955)
  - No operating systems
  - Assemblers and loaders were almost the only system software provided
Historical Overview of Operating Systems Development (continued)

- Second generation of system software (1955–1965)
  - Batch operating systems
  - Ran collections of input programs one after the other
  - Included a command language
Figure 6.18
Operation of a Batch Computer System
Third-generation operating systems (1965–1985)

- Multiprogrammed operating systems
  - Permitted multiple user programs to run at once
- Time sharing
  - Use of *time slices* to service multiple users on the same computer
Fourth-generation operating systems (1985–present)

- Network operating systems

- Virtual environment treats resources physically residing on the computer in the same way as resources available through the computer’s network
Figure 6.22
The Virtual Environment Created by a Network Operating System
The Future

- Operating systems will continue to evolve
- Possible characteristics of fifth-generation systems
  - Multimedia user interfaces
  - Parallel processing systems
  - Completely distributed computing environments
Figure 6.23
Structure of a Distributed System
<table>
<thead>
<tr>
<th>Generation</th>
<th>Approximate Dates</th>
<th>Major Advances</th>
</tr>
</thead>
</table>
| First      | 1945–1955         | No operating system available  
                                                   Programmers operated the machine themselves |
| Second     | 1955–1965         | Batch operating systems  
                                                   Improved system utilization  
                                                   Development of the first command language |
| Third      | 1965–1985         | Multiprogrammed operating systems  
                                                   Time-sharing operating systems  
                                                   Increasing concern for protecting programs from damage by other programs  
                                                   Creation of privileged instructions and user instructions  
                                                   Interactive use of computers  
                                                   Increasing concern for security and access control  
                                                   First personal computer operating systems |
| Fourth     | 1985–present      | Network operating systems  
                                                   Client-server computing  
                                                   Remote access to resources  
                                                   Graphical user interfaces  
                                                   Real-time operating systems  
                                                   Embedded systems |
| Fifth      | ??                | Multimedia user interfaces  
                                                   Massively parallel operating systems  
                                                   Distributed computing environments |

Figure 6.24
Some of the Major Advances in Operating Systems Development
Terminology

GUI - Graphical User Interface OS
- Has the capability of using a mouse and emphasizes visual devices such as icons. Examples: System X, newer UNIX versions, Linux, Windows XP (and 95, 98, CE, NT 4.0, 2000)

Multi -User OS
- Multiple users use the computer and run programs at the same time. Examples: All of the above except Windows CE. Special cases include:
  - Timesharing OS - Use of time slices to service multiple users in the same computer.
  - Distributed OS-- Computers distributed geographically can operate separately or together.

Multitasking OS
- Allow multiple software processes to be run at the same time. Examples: System X, UNIX, Windows XP (and 95, 98, NT 4.0, 2000)

Multithreading OS
- Allow different parts of a software program to run concurrently. Examples: UNIX, Windows XP (and 95, 98, NT 4.0, 2000)

Multiprocessing OS
- Allows multiple processors to be utilized as one machine. Examples: UNIX, Windows XP, Windows 2000, Windows NT 4.0

Batch system OS-
- Jobs are bundled together with the instructions necessary to allow them to be processed without intervention. Often jobs of a similar nature can be bundled together to further increase economy.
- This is an older type of operating system. Today, on large systems, jobs can be batched, but you don't see OS that are strictly batch systems anymore.

Real-time OS-
- Jobs must operate in a timely manner while a user interacts with the operating system.
Summary

- System software acts as an intermediary between the users and the hardware.

- Assembly language creates a more productive, user-oriented environment than machine language.

- An assembler translates an assembly language program into a machine language program.
Summary

- Responsibilities of the operating system
  - User interface management
  - Program scheduling and activation
  - Control of access to system and files
  - Efficient resource allocation
  - Deadlock detection and error detection