System Software

SYSTEM SOFTWARE: AN OVERVIEW
ASSEMBLERS AND ASSEMBLY LANGUAGE
OPERATING SYSTEMS
IMPLEMENTING SYSTEM SOFTWARE
System Software: An Overview

- “Naked” computer hard to deal with, e.g.,
  1. Write machine language program.
  2. Load program into memory starting at address 0.
  3. Load 0 into PC and start execution.

- Need virtual machine interface, which does the following:
  - Hides details of machine operation.
  - Does not require in-depth knowledge of machine internals.
  - Provides easy access to system resources.
  - Prevents accidental or intentional damage to hardware, programs, and data.

- Create virtual machine and associated interface with system software.
Figure 6.1 The Role of System Software
System Software: An Overview (Cont’d)

- System software provided by **Operating System (OS)**.
- Many types of system software in an OS, e.g.,
  - **Graphical User Interface (GUI)**: Access system services.
  - **Language services**: Allow programming in high-level languages, e.g., text editor, assembler, loader, compiler, debugger.
  - **Memory manager**: Allocate memory for programs and data and retrieve memory after use.
  - **Information manager**: Organize program and data files for easy access, e.g., folders, directories.
  - **I/O system manager**: Access I/O devices.
  - **Scheduler**: Manage multiple active programs.
System Software: An Overview (Cont’d)

Figure 6.2
Types of System Software
OS dramatically simplifies creation of software, e.g.,

1. Write **source program** $P$ in high-level programming language using a text editor.

2. Use an information manager to store $P$ as a file in a directory.

3. Use a compiler and an assembler to translate $P$ into an equivalent machine language program $M$.

4. Use scheduler to load, schedule, and run $M$ (with scheduler calling memory manager and loader).

5. Use I/O system manager to display output on screen.

6. If necessary, use debugger to isolate and text editor to correct program errors.
Assemblers and Assembly Language: Overview

- An assembly language is the human-friendly version of a machine language, courtesy of several features:
  - Symbolic op-codes, e.g., ADD, COMPARE;
  - Symbolic memory addresses and labels, e.g., IND, ONE, AFTERLOOP; and
  - **Pseudo-ops** which specify extra assembler directives, e.g., .DATA, .BEGIN, .END.

- An assembler converts an assembly language source program into a machine language *object program*; a loader then places the instructions in that object program in the specified memory addresses.
Assemblers and Assembly Language: Overview (Cont’d)

Figure 6.3
The Continuum of Programming Languages
Assemblers and Assembly Language: Overview (Cont’d)

Figure 6.4
The Translation/Loading/Execution Process (Assembly --> M.C.)
Assemblers and Assembly Language: An Example Assembly Language

<table>
<thead>
<tr>
<th>OC</th>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOAD Lbl</td>
<td>$CON(Lbl) \rightarrow R$</td>
</tr>
<tr>
<td>1</td>
<td>STORE Lbl</td>
<td>$R \rightarrow CON(Lbl)$</td>
</tr>
<tr>
<td>2</td>
<td>CLEAR Lbl</td>
<td>$0 \rightarrow CON(Lbl)$</td>
</tr>
<tr>
<td>3</td>
<td>ADD Lbl</td>
<td>$R + CON(Lbl) \rightarrow R$</td>
</tr>
<tr>
<td>4</td>
<td>INCREDENT Lbl</td>
<td>$CON(Lbl) + 1 \rightarrow CON(Lbl)$</td>
</tr>
<tr>
<td>5</td>
<td>SUBTRACT Lbl</td>
<td>$R - CON(Lbl) \rightarrow R$</td>
</tr>
<tr>
<td>6</td>
<td>DECREMENT Lbl</td>
<td>$CON(Lbl) - 1 \rightarrow CON(Lbl)$</td>
</tr>
<tr>
<td>7</td>
<td>COMPARE Lbl</td>
<td>if $CON(Lbl) &gt; R$ then $GT = 1$ else 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $CON(Lbl) = R$ then $EQ = 1$ else 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $CON(Lbl) &lt; R$ then $LT = 1$ else 0</td>
</tr>
<tr>
<td>8</td>
<td>JUMP Lbl</td>
<td>$ADDR(Lbl) \rightarrow PC$</td>
</tr>
<tr>
<td>9</td>
<td>JUMPGT Lbl</td>
<td>if $GT = 1$ then $ADDR(Lbl) \rightarrow PC$</td>
</tr>
</tbody>
</table>
### Assemblers and Assembly Language: An Example Assembly Language (Cont’d)

<table>
<thead>
<tr>
<th>OC</th>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>JUMPEQ Lbl</td>
<td>if $EQ = 1$ then $ADDR(Lbl) \rightarrow PC$</td>
</tr>
<tr>
<td>11</td>
<td>JUMPLT Lbl</td>
<td>if $LT = 1$ then $ADDR(Lbl) \rightarrow PC$</td>
</tr>
<tr>
<td>12</td>
<td>JUMPNEQ Lbl</td>
<td>if $EQ = 0$ then $ADDR(Lbl) \rightarrow PC$</td>
</tr>
<tr>
<td>13</td>
<td>IN Lbl</td>
<td>Store input value at $ADDR(Lbl)$</td>
</tr>
<tr>
<td>14</td>
<td>OUT Lbl</td>
<td>Output $CON(Lbl)$</td>
</tr>
<tr>
<td>15</td>
<td>HALT</td>
<td>Stop program execution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pseudo-op</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DATA Val</td>
<td>Create memory cell with value $Val$</td>
</tr>
<tr>
<td>.BEGIN</td>
<td>Begin program translation process</td>
</tr>
<tr>
<td>.END</td>
<td>Begin program translation process</td>
</tr>
</tbody>
</table>
Assemblers and Assembly Language: An Example Assembly Language (Cont’d)

- Access .DATA-created values with symbolic labels, e.g.,

\[
\text{NEGSEVEN: .DATA } -7
\]

\[
\downarrow
\]

\[
\begin{array}{c}
54: \\
10000111
\end{array}
\]

\[
\text{NEGSEVEN} = 54
\]

- To prevent .DATA-created values from being interpreted as instructions, place all .DATA pseudo-ops after HALT at the end of the program.
Assemblers and Assembly Language:  
Example Assembly Language Code

set $A$ to the value of $B + C$

```
LOAD B
ADD C
STORE A
...
```

```
A: .DATA 1
B: .DATA 2
C: .DATA 3
```
Assemblers and Assembly Language: Example Assembly Language Code (Cont’d)

if $A > B$ then
  set $C$ to the value of $A$
else
  set $C$ to the value of $B$

LOAD B
COMPARE A
JUMPGT IFPART
LOAD B
STORE C
JUMP ENDFIF

IFPART:
  LOAD A
  STORE C

ENDIF:
  •••
  •••

A: .DATA 1
B: .DATA 2
C: .DATA 3
Assemblers and Assembly Language:
Example Assembly Language Code (Cont’d)

set \( IND \) to 0
while \( IND \leq MAXIND \) do
  \( \langle LOOPBODY \rangle \)
  set \( IND \) to \( IND + 1 \)
  \( \langle LOOPBODY \rangle \)
  JUMP \( LOOPSTART \)
  \( \langle LOOPBODY \rangle \)
  INCREMENT \( IND \)
  JUMP \( LOOPSTART \)

\( LOOPSTART: \)
  LOAD \( MAXIND \)
  COMPARE \( IND \)
  JUMPGT \( LOOPEND \)

\( LOOPEND: \)
  \( \ldots \)
  \( \ldots \)

\( ZERO: \)
  .DATA 0

\( IND: \)
  .DATA 0

\( MAXIND: \)
  .DATA 25
Assemblers and Assembly Language: An Assembly Language Program

Consider the following algorithm for computing and printing the sum of all values in a −1-terminated list:

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Set $SUM$ to 0</td>
</tr>
<tr>
<td>2.</td>
<td>Read the first list value into $CURVAL$</td>
</tr>
<tr>
<td>3.</td>
<td>while ($CURVAL \neq -1$) do</td>
</tr>
<tr>
<td>4.</td>
<td>Set $SUM$ to $SUM + CURVAL$</td>
</tr>
<tr>
<td>5.</td>
<td>Read the next list value into $CURVAL$</td>
</tr>
<tr>
<td>6.</td>
<td>Print the value of $SUM$</td>
</tr>
<tr>
<td>7.</td>
<td>Stop</td>
</tr>
</tbody>
</table>

Let’s implement this algorithm in assembly language.
Assemblers and Assembly Language: An Assembly Language Program (Cont’d)

Step 1
SUM: .DATA 0
CURVAL: .DATA 0
ENDVAL: .DATA -1
.END

Step 2
.IN CURVAL

Step 3
LOOPSTART:
LOAD ENDVAL
COMPARE CURVAL
JUMPEQ LOOPEND

Step 4
LOAD SUM
ADD CURVAL
STORE SUM

Step 5
IN CURVAL
JUMP LOOPSTART

Step 6
LOOPEND:
OUT SUM

Step 7
HALT
Assemblers and Assembly Language: The Assembly Process

- Duties of the assembler:
  1. Translate symbolic op-codes into binary.
  2. Translate symbolic addresses and labels into binary.
  3. Execute all pseudo-ops.
  4. Place translation in object program file.

- As symbolic addresses and labels may be used before they are defined, translation done in two passes:
  
  **Pass 1**: Accumulate all symbolic label / binary address bindings in symbol table.
  
  **Pass 2**: Resolve all symbolic label references.

- Op-code / symbolic label lookup typically optimized by alphabetic op-code / label sorting and binary search.
Assemblers and Assembly Language: The Assembly Process (Cont’d)

<table>
<thead>
<tr>
<th>Label</th>
<th>Code</th>
<th>Location Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP:</td>
<td>IN X</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>IN Y</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>LOAD X</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>COMPARE Y</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>JUMPPGT DONE</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>OUT X</td>
<td>5</td>
</tr>
<tr>
<td>DONE:</td>
<td>JUMP LOOP</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>OUT Y</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>HALT</td>
<td>8</td>
</tr>
<tr>
<td>X:</td>
<td>.DATA 0</td>
<td>9</td>
</tr>
<tr>
<td>Y:</td>
<td>.DATA 0</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>DONE</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.10 Generation of the Symbol Table
Assemblers and Assembly Language: The Assembly Process (Cont’d)

### Instruction Format:
- **OP Code**: 4 bits
- **Address**: 12 bits

### Object Program:

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Language Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1101 00000000001001</td>
<td>IN X</td>
</tr>
<tr>
<td>0001</td>
<td>1101 00000000001010</td>
<td>IN Y</td>
</tr>
<tr>
<td>0010</td>
<td>0000 00000000001001</td>
<td>LOAD X</td>
</tr>
<tr>
<td>0011</td>
<td>0111 00000000001010</td>
<td>COMPARE Y</td>
</tr>
<tr>
<td>0100</td>
<td>1001 0000000000111</td>
<td>JUMPGT DONE</td>
</tr>
<tr>
<td>0101</td>
<td>1110 00000000001001</td>
<td>OUT X</td>
</tr>
<tr>
<td>0110</td>
<td>1000 000000000000000</td>
<td>JUMP LOOP</td>
</tr>
<tr>
<td>0111</td>
<td>1110 00000000001010</td>
<td>OUT Y</td>
</tr>
<tr>
<td>1000</td>
<td>1111 0000000000000000</td>
<td>HALT</td>
</tr>
<tr>
<td>1001</td>
<td>0000 0000000000000000</td>
<td>The constant 0</td>
</tr>
<tr>
<td>1010</td>
<td>0000 0000000000000000</td>
<td>The constant 0</td>
</tr>
</tbody>
</table>

**Figure 6.13** Example of an Object Program
Operating Systems

Major duties of an operating system:

- **User Interface**: Accept *system commands* from user and, if these commands are valid, schedule appropriate system software to execute command.

- **System Security and Protection**: Determine valid users and valid activities and accesses for users using usernames, passwords, and *access control lists*.

- **Efficient Management of Resources**: Optimize processor use by maintaining Running (active program), Ready (programs ready to execute), and Waiting (programs waiting on I/O requests) queues.

- **The Safe Use of Resources**: Prevent *deadlock* (two or more users have partial required resources) using resolution algorithms and protocols.
Implementing System Software: Compilers

- First compilers pioneered by Grace Hopper in early 1950s.
- Compilers can be cascaded, e.g., translate high-level language into assembler language and assembler language into machine language.

Grace Hopper (1906–1992)
Implementing System Software: Programming Languages

- FORTRAN (FORmula TRANslation) created by Backus team at IBM in 1957; designed for scientific computation.
- COBOL (COnmon Business-Oriented Language) created by industry / government committee in 1959.

John Backus (1924–2007)

Grace Hopper teaching COBOL (early 1960’s)
Implementing System Software: Programming Languages (Cont’d)

• BASIC (Beginner’s All-purpose Symbolic Instruction Code) created by Thomas Kurtz (1928–) and John Kemeney (1926-1992) at Dartmouth College in 1964.

• Designed as a programming language for everyone.
Implementing System Software: Operating Systems

- OS only possible after sufficient computer memory available for system software starting around 1955.
- Three OS generations to date:
     Run multiple programs in sequence with aid of Job Control Language (JCL).
     Run multiple programs in apparent parallel by swapping programs in and out of the control unit.
- Future OS will incorporate multimedia user interfaces (e.g., voice / gesture-based) and fully distributed execution.
Implementing System Software: User Interfaces

Doug Engelbart (1925-2013)

Computer Mouse (1965)

- Engelbart and colleagues develop graphical user interface (GUI) and computer mouse at Stanford starting in 1963.
Implementing System Software: User Interfaces (Cont’d)

“The Mother of All Demos” (1968)
Implementing System Software: User Interfaces (Cont’d)

Xerox Alto (1973) [$25K (est)]

- Alto was first modern GUI-driven PC; also incorporated local-area networking and laserjet printers (WYSIWYG).

Xerox Star (1981) [$75K]

- Star intended for use in large corporations.
Implementing System Software: User Interfaces (Cont’d)

XEROX 8010 Star Information System

Star provides integrated text and graphics. A variety of type sizes and styles may be used.

<table>
<thead>
<tr>
<th>Description</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pear</td>
<td>$0.25</td>
</tr>
<tr>
<td>Beans</td>
<td>$0.50</td>
</tr>
</tbody>
</table>

This is some text in a text frame.

Field

Button
Starting in 1979, Steve Jobs re-creates GUI-based functionality at Apple in the Lisa and Macintosh PCs.

Part of Macintosh application and OS development subcontracted to Microsoft starting in 1981.
Implementing System Software: User Interfaces (Cont’d)

- Microsoft releases Windows v1.0 in 1985; legally emulated portions of Lisa and Mac look.
- Microsoft releases Windows v2.0 in late 1987; is not only much faster but (now illegally) identical to Mac look.
- By late 1980s, Windows has 90% market-share in GUI-based PC computing.
And If You Liked This . . .

- MUN Computer Science courses on this area:
  - COMP 2001: Object-oriented Programming and HCI
  - COMP 2003: Operating Systems
  - COMP 4712: Compiler Construction

- MUN Computer Science professors teaching courses / doing research in in this area:
  - Ed Brown
  - Rod Byrne