Computer Science 1000: Part #6

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**.

System Software: An Overview (Cont'd)

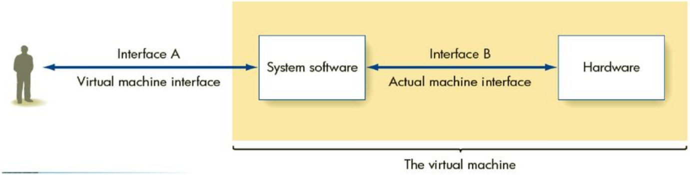


Figure 6.1 The Role of System Software

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)

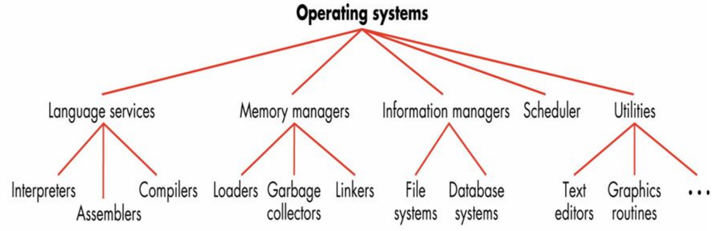


Figure 6.2

Types of System Software

System Software: An Overview (Cont'd)

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)

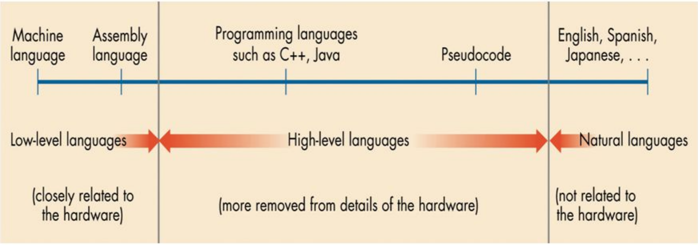


Figure 6.3
The Continuum of Programming Languages

Assemblers and Assembly Language: Overview (Cont'd)

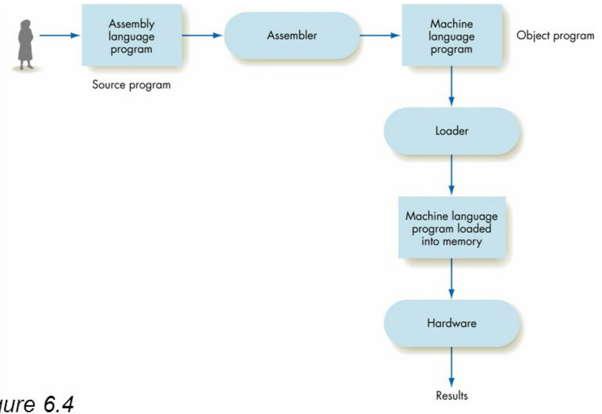


Figure 6.4

The Translation/Loading/Execution Process (Assembly --> M.C.)

Assemblers and Assembly Language: An Example Assembly Language

OC	Instruction	Meaning
0	`LOAD Lbl`	$CON(Lbl) \longrightarrow R$
1	`STORE Lbl`	$R \longrightarrow CON(Lbl)$
2	`CLEAR Lbl`	$0 \longrightarrow CON(Lbl)$
3	`ADD Lbl`	$R + CON(Lbl) \longrightarrow R$
4	`INCREMENT Lbl`	$CON(Lbl) + 1 \longrightarrow CON(Lbl)$
5	`SUBTRACT Lbl`	$R - CON(Lbl) \longrightarrow R$
6	`DECREMENT Lbl`	$CON(Lbl) - 1 \longrightarrow CON(Lbl)$
7	`COMPARE Lbl`	if $CON(Lbl) > R$ then $GT = 1$ else 0
		if $CON(Lbl) = R$ then $EQ = 1$ else 0
		if $CON(Lbl) < R$ then $LT = 1$ else 0
8	`JUMP Lbl`	$ADDR(Lbl) \longrightarrow PC$
9	`JUMPGT Lbl`	if $GT = 1$ then $ADDR(Lbl) \longrightarrow PC$

Assemblers and Assembly Language: An Example Assembly Language (Cont'd)

OC	Instruction	Meaning
10	`JUMPEQ Lbl`	if $EQ = 1$ then $ADDR(Lbl) \longrightarrow PC$
11	`JUMPLT Lbl`	if $LT = 1$ then $ADDR(Lbl) \longrightarrow PC$
12	`JUMPNEQ Lbl`	if $EQ = 0$ then $ADDR(Lbl) \longrightarrow PC$
13	`IN Lbl`	Store input value at $ADDR(Lbl)$
14	`OUT Lbl`	Output $CON(Lbl)$
15	`HALT`	Stop program execution

Pseudo-op	Meaning
`.DATA Val`	Create memory cell with value *Val*
`.BEGIN`	Begin program translation process
`.END`	Begin program translation process

Assemblers and Assembly Language: An Example Assembly Language (Cont'd)

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

```
NEGSEVEN:  .DATA -7
```

$$\Downarrow$$

54: | 10000111 |

$$\texttt{NEGSEVEN} = 54$$

- 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 ENDIF
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 $IND + 1$

```
            LOAD ZERO
            STORE IND
LOOPSTART:  LOAD MAXIND
            COMPARE IND
            JUMPGT LOOPEND
            ⟨LOOPBODY⟩
            INCREMENT IND
            JUMP LOOPSTART
  LOOPEND:  ...
            ...
     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:

Step	Operation
1.	Set SUM to 0
2.	Read the first list value into $CURVAL$
3.	while ($CURVAL \neq -1$) do
4.	Set SUM to $SUM + CURVAL$
5.	Read the next list value into $CURVAL$
6.	Print the value of SUM
7.	Stop

Let's implement this algorithm in assembly language.

Assemblers and Assembly Language: An Assembly Language Program (Cont'd)

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

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)

LABEL	CODE		LOCATION COUNTER
LOOP:	IN	X	0
	IN	Y	1
	LOAD	X	2
	COMPARE	Y	3
	JUMPGT	DONE	4
	OUT	X	5
	JUMP	LOOP	6
DONE:	OUT	Y	7
	HALT		8
X:	.DATA	0	9
Y:	.DATA	0	10

(a)

SYMBOL TABLE

SYMBOL	ADDRESS VALUE
LOOP	0
DONE	7
X	9
Y	10

(b)

Figure 6.10 Generation of the Symbol Table

Assemblers and Assembly Language: The Assembly Process (Cont'd)

INSTRUCTION FORMAT:	OP CODE	ADDRESS
	4 bits	12 bits

OBJECT PROGRAM:

Address	*Machine Language Instruction*	*Meaning*
0000	1101 000000001001	IN X
0001	1101 000000001010	IN Y
0010	0000 000000001001	LOAD X
0011	0111 000000001010	COMPARE Y
0100	1001 000000000111	JUMPGT DONE
0101	1110 000000001001	OUT X
0110	1000 000000000000	JUMP LOOP
0111	1110 000000001010	OUT Y
1000	1111 000000000000	HALT
1001	0000 000000000000	The constant 0
1010	0000 000000000000	The constant 0

Figure 6.13 Example of an Object Program

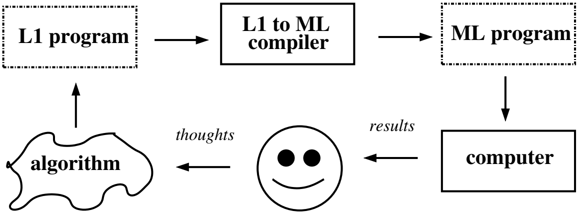
Operating Systems

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

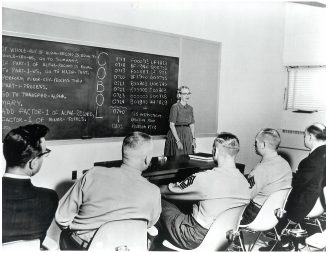
- First compilers pioneered by Grace Hopper in early 1950s.
- Compilers can be cascaded, *e.g.*, translate high-level language into assembler language and assembler language into machine language.



Grace Hopper (1906–1992)

Implementing System Software: Programming Languages

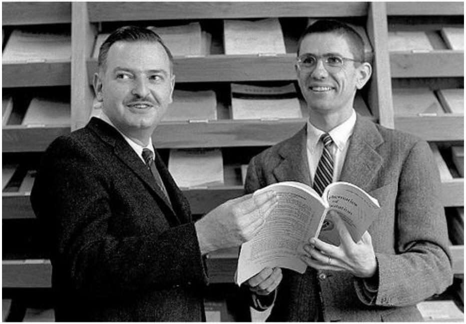
John Backus (1924–2007)



Grace Hopper teaching COBOL (early 1960's)

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

Implementing System Software: Programming Languages (Cont'd)

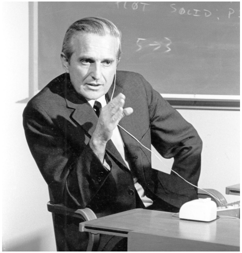


- BASIC (Beginner's All-purpose Symbolic Instruction Code) created by Thomas Kurtz (1928–) and John Kemeney (1926-1992) at Dartmouth College in 1964.
- Designed as a programming language for *everyone*.

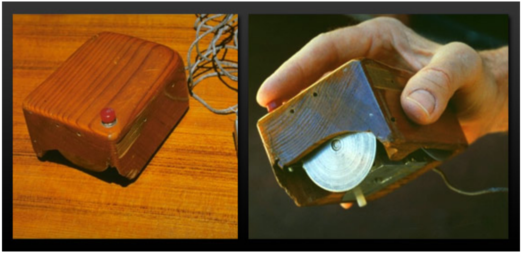
Implementing System Software: Operating Systems

- OS only possible after sufficient computer memory available for system software starting around 1955.
- Three OS generations to date:
  1. Single-user batch-style OS (1955–1965)
     Run multiple programs in sequence with aid of Job Control Language (JCL).
  2. Multi-user time-sharing OS (1965–1985)]
     Run multiple programs in apparent parallel by swapping programs in and out of the control unit.
  3. Multi-user network OS (1985–present)
- 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)

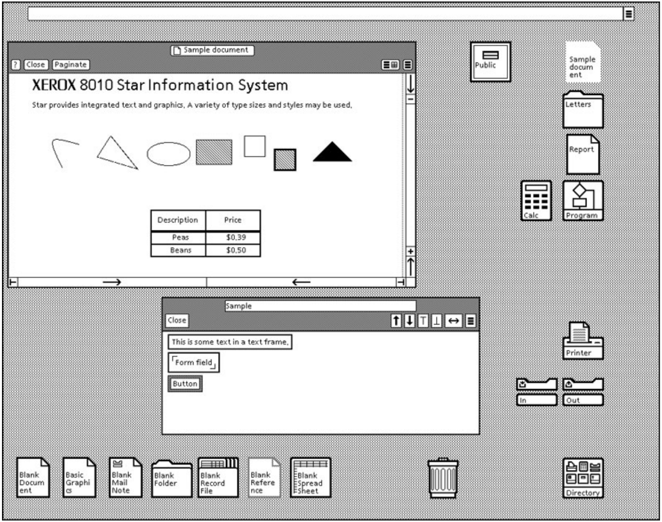
Implementing System Software: User Interfaces (Cont'd)

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

Xerox Star (1981) [$75K]

- Alto was first modern GUI-driven PC; also incorporated local-area networking and laserjet printers (WYSIWYG).
- Star intended for use in large corporations.

Implementing System Software: User Interfaces (Cont'd)

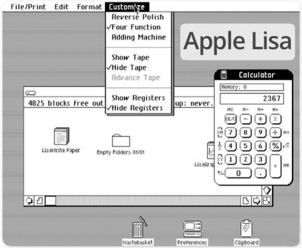


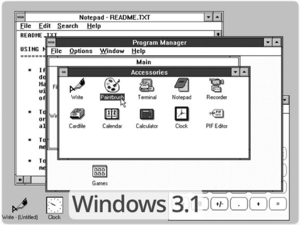
Implementing System Software: User Interfaces (Cont'd)

Apple Macintosh (1984) [$2,500]

- 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 sub-contracted 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.
- Apple sues Microsoft over Windows 2.0 "look and feel" in 1988; case dismissed in 1991.
- 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
  - Ashoke Deb
  - Wlodek Zuberek