Chapter 12:

Operating System

Usage and Copyright Notice:

Copyright 2005 © Noam Nisan and Shimon Schocken


The book web site, www.idc.ac.il/tecs, features 13 such presentations, one for each book chapter. Each presentation is designed to support about 3 hours of classroom or self-study instruction.

You are welcome to use or edit this presentation for instructional and non-commercial purposes.

If you use our materials, we will appreciate it if you will include in them a reference to the book’s web site.

And, if you have any comments, you can reach us at tecs.ta@gmail.com
Where we are at:

**Abstract design**
  - Chapters 10 - 11
- VM Translator
  - Chapters 7 - 8
- Assembler
  - Chapter 6

**Software hierarchy**
- Compiler
- Virtual Machine
  - abstract interface
- Assembly Language
  - abstract interface

**Hardware hierarchy**
- Machine Language
- Computer Architecture
  - abstract interface
  - Chapters 4 - 5
- Hardware Platform
  - abstract interface
  - Chapters 1 - 3
- Chips & Logic Gates
- Electrical Engineering

**Human Thought**
- Abstract design
  - Chapters 9, 12

/** Computes the average of a sequence of integers. */
class Main {
    function void main() {
        var Array a;
        var int length;
        var int i, sum;

        let length = Keyboard.readInt("How many numbers? ");
        let a = Array.new(length); // Constructs the array
        let i = 0;

        while (i < length) {
            let a[i] = Keyboard.readInt("Enter the next number: ");
            let sum = sum + a[i];
            let i = i + 1;
        }

        do Output.printString("The average is: ");
        do Output.printInt(sum / length);
        do Output.println();
        return;
    }
}

/** Computes the average of a sequence of integers. */
class Main {
    function void main() {
        var Array a;
        var int length;
        var int i, sum;

        let length = Keyboard.readInt("How many numbers? ");
        let a = Array.new(length); // Constructs the array
        let i = 0;

        while (i < length) {
            let a[i] = Keyboard.readInt("Enter the next number: ");
            let sum = sum + a[i];
            let i = i + 1;
        }

        do Output.printString("The average is: ");
        do Output/printInt(sum / length);
        do Output.println();
        return;
    }
}
Typical OS functions

<table>
<thead>
<tr>
<th>Language extensions / standard library</th>
<th>System-oriented services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mathematical operations</td>
<td>Memory management</td>
</tr>
<tr>
<td>(abs, sqrt, ...)</td>
<td>(objects, arrays, ...)</td>
</tr>
<tr>
<td>Abstract data types</td>
<td>I/O device drivers</td>
</tr>
<tr>
<td>(String, Date, ...)</td>
<td>Mass storage</td>
</tr>
<tr>
<td>Output functions</td>
<td>File system</td>
</tr>
<tr>
<td>(printChar, printString ...)</td>
<td>Multi-tasking</td>
</tr>
<tr>
<td>Input functions</td>
<td>UI management (shell / windows)</td>
</tr>
<tr>
<td>(readChar, readLine ...)</td>
<td>Security</td>
</tr>
<tr>
<td>Graphics functions</td>
<td>Communications</td>
</tr>
<tr>
<td>(drawPixel, drawCircle, ...)</td>
<td>And more ...</td>
</tr>
<tr>
<td>And more ...</td>
<td>And more ...</td>
</tr>
</tbody>
</table>
The Jack OS

- **Math**: Provides basic mathematical operations;
- **String**: Implements the String type and string-related operations;
- **Array**: Implements the Array type and array-related operations;
- **Output**: Handles text output to the screen;
- **Screen**: Handles graphic output to the screen;
- **Keyboard**: Handles user input from the keyboard;
- **Memory**: Handles memory operations;
- **Sys**: Provides some execution-related services.
Jack OS API

class Math {
    function void init()
    function int abs(int x)
    function int multiply(int x, int y)
    function int divide(int x, int y)
    function int min(int x, int y)
    function int max(int x, int y)
    function int sqrt(int x)
}

class String {
    constructor String new(int maxLength)
    method void dispose()
    method int length()
    method char charAt(int j)
    method void setCharAt(int j, char c)
    method String appendChar(char c)
    method void eraseLastChar()
    method int intValue()
    method void setInt(int j)
    function char backSpace()
    function char doubleQuote()
    function char newLine()
}

Class Array {
    function Array new(int size)
    method void dispose()
}

class Output {
    function void moveCursor(int i, int j)
    function void printChar(char c)
    function void printString(String s)
    function void printInt(int i)
    function void println()
    function void backSpace()
}

Class Screen {
    function void clearScreen()
    function void setColor(boolean b)
    function void drawPixel(int x, int y)
    function void drawLine(int x1, int y1, int x2, int y2)
    function void drawRectangle(int x1, int y1, int x2, int y2)
    function void drawCircle(int x, int y, int r)
}

class Memory {
    function int peek(int address)
    function void poke(int address, int value)
    function Array alloc(int size)
    function void deAlloc(Array o)
}

class Keyboard {
    function char keyPressed()
    function char readChar()
    function String readLine(String message)
    function int readInt(String message)
}

class Sys {
    function void halt()
    function void error(int errorCode)
    function void wait(int duration)
}
A typical OS:

- Is modular and scalable
- Empowers programmers (language extensions)
- Empowers users (file system, GUI, ...)
- Closes gaps between software and hardware
- Runs in “protected mode”
- Typically written in some high level language
- Typically grow gradually, assuming more and more functions
- Must be efficient.
Efficiency first

- We have to implement various operations on \( n \)-bit binary numbers \((n = 16, 32, 64, \ldots)\). Example: multiplication

- Naïve algorithm: to multiply \( x \times y \): \{" for \( i = 1 \ldots y \) do \( \text{sum} = \text{sum} + x \} \)

  Run-time is proportional to \( y \)

  In a 64-bit system, \( y \) can be as large as \( 2^{64} \).

  The multiplication will take years to complete

  If the run-time were proportional to 64 instead, we are OK

- In general, algorithms that operate on \( n \)-bit inputs are either:
  - **Naïve**: run-time is proportional to the value of the \( n \)-bit inputs
  - **Good**: run-time is proportional to \( n \).
Example I: multiplication

The “steps”

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
1 & 0 & 1 & \\
1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 = 5 & 5
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
1 & 0 & 1 & \\
1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 = 5 & 5
\end{array}
\]

The algorithm explained (first 4 of 16 iteration)

\[
x: \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1
\]
\[
y: \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1
\]
\[
j^{th} \text{ bit of } y
\]
\[
0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1
\]
\[
0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0
\]
\[
0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1
\]
\[
1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0
\]
\[
x \cdot y: \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad \text{sum}
\]

\textbf{multiply}(x, y):

// Where \( x, y \geq 0 \)

\[\text{sum} = 0\]

\[\text{shiftedX} = x\]

for \( j = 0 \ldots (n - 1) \) do

if (j-th bit of y) = 1 then

\[\text{sum} = \text{sum} + \text{shiftedX}\]

\[\text{shiftedX} = \text{shiftedX} \times 2\]

- Run-time: proportional to \( n \)
- Can be implemented in SW or HW
- Division: similar idea.
Example II: square root

- The square root function has two useful properties:
  - An inverse function that we know how to compute
  - Monotonically increasing

- Ergo, square root can be computed via binary search:

```c
sqrt(x):

// Compute the integer part of \( y = \sqrt{x} \). Strategy:
// Find an integer \( y \) such that \( y^2 \leq x < (y+1)^2 \) (for \( 0 \leq x < 2^x \))
// By performing a binary search in the range 0 ... \( 2^{x/2} - 1 \).

y = 0
for j = n/2 - 1 ... 0 do
    if \((y+2^j)^2 \leq x\) then \( y = y + 2^j \)
return y
```

- Number of loop iterations is bound by \( n/2 \), thus the run-time is \( O(n) \).
Math operations (in the Jack OS)

class Math {
    function void init()
    function int abs(int x)
    \[ \checkmark \text{function int multiply(int x, int y)} \]
    \[ \checkmark \text{function int divide(int x, int y)} \]
    function int min(int x, int y)
    function int max(int x, int y)
    \[ \checkmark \text{function int sqrt(int x)} \]
}
String processing (in the Jack OS))

Class String {
    constructor String new(int maxLength)
    method void dispose()
    method int length()
    method char charAt(int j)
    method void setCharAt(int j, char c)
    method String appendChar(char c)
    method void eraseLastChar()
    method int intValue()
    method void setInt(int j)
    function char backSpace()
    function char doubleQuote()
    function char newLine()
}

Converting a single digit to its ASCII code

- \( \text{ASCIICode}(\text{digit}) = 48 + \text{digit} \)
- Reverse conversion: easy.
Converting a number to a string

- SingleDigit-to-character conversions: done
- Number-to-string conversions:

```cpp
// Convert a non-negative number to a string
int2String(n):
    lastDigit = n % 10
    c = character representing lastDigit
    if n < 10
        return c (as a string)
    else
        return int2String(n / 10).append(c)

// Convert a string to a non-negative number
string2Int(s):
    v = 0
    for i = 1... length of s do
        d = integer value of the digit s[i]
        v = v * 10 + d
    return v

// (Assuming that s[1] is the most significant digit character of s.)
```
Memory management (in the Jack OS)

```java
class Memory {
    function int peek(int address)
    function void poke(int address, int value)
    function Array alloc(int size)
    function void deAlloc(Array o)
}
```
Memory management (simple)

- When a program constructs (destructs) an object, the OS has to allocate (de-allocate) a RAM block on the heap:
  
  - `alloc(size)`: returns a reference to a free RAM block of size `size`
  - `deAlloc(object)`: recycles the RAM block that `object` points at

---

**Initialization:** \( \text{free} = \text{heapBase} \)

// Allocate a memory block of size words.

```c
alloc(size):
    pointer = free
    free = free + size
    return pointer
```

// De-allocate the memory space of a given object.

```c
deAlloc(object):
do nothing
```

- The data structure that this algorithm manages is a single pointer: `free`.
Memory management (improved)

Initialization:

```c
.freeList = heapBase
.freeList.length = heapLength
.freeList.next = null
```

// Allocate a memory space of size words.

```c
alloc(size):
    Search .freeList using best-fit or first-fit heuristics
to obtain a segment with .segment.length > size
If no such segment is found, return failure
    (or attempt defragmentation)
    block = needed part of the found segment
    (or all of it, if the segment remainder is too small)
    Update .freeList to reflect the allocation
    block[-1] = size + 1    // Remember block size, for de-allocation
    Return block
```

// Deallocate a decommissioned object.

deAlloc(object):

```c
    segment = object - 1
    segment.length = object[-1]
    Insert segment into the .freeList
```
Peek and poke

```java
class Memory {
    function int peek(int address)
    function void poke(int address, int value)
    function Array alloc(int size)
    function void deAlloc(Array o)
}
```

- Implementation: exploiting exotic casting in Jack:

```java
// To create a Jack-level "proxy" of the RAM:
var Array memory;
let memory = 0;
// From this point on we can use code like:
let x = memory[j] // Where j is any RAM address
let memory[j] = y // Where j is any RAM address
```
Graphics primitives (in the Jack OS)

Class Screen {

function void clearScreen()

function void setColor(boolean b)

function void drawPixel(int x, int y)

function void drawLine(int x1, int y1, int x2, int y2)

function void drawRectangle(int x1, int y1, int x2, int y2)

function void drawCircle(int x, int y, int r)

}

Memory-mapped screen
Pixel drawing

```
drawPixel (x, y):
   // Hardware-specific.
   // Assuming a memory mapped screen:
   Write a predetermined value in the RAM location corresponding to screen location (x, y).
```

- **Implementation:** using `poe(address, value)`
Line drawing

\begin{align*}
\text{drawLine}(x, y, x+dx, y+dy): \\
// \text{Assuming } dx, dy > 0 \\
\text{initialize } (a, b) = (0,0) \\
\text{while } a \leq dx \text{ and } b \leq dy \text{ do} \\
\quad \text{drawPixel}(x+a, y+b) \\
\quad \text{if } a/dx < b/dy \text{ then } a++ \text{ else } b++
\end{align*}

- $dx=0$ and $dy=0$ are not handled
- Must also handle $(dx,dy<0)$, $(dx>0,dy<0)$, $(dx<0,dy>0)$
Line drawing

```plaintext
drawLine(x, y, x+dx, y+dy):
    // Assuming dx,dy > 0
    initialize (a,b) = (0,0)
    while a ≤ dx and b ≤ dy do
        drawPixel(x+a, y+b)
        if a/dx < b/dy then a++ else b++
```

\(a/dx < b/dy\) is the same as \(a*dy < b*dx\)

// To test whether \(a/dx < b/dy\), maintain a variable adyMinusbdx,
// and test if it becomes negative.
Initialization: set adyMinusbdx = 0
When a++ is performed: set adyMinusbdx = adyMinusbdx + dy
When b++ is performed: set adyMinusbdx = adyMinusbdx - dx

And so we are back to addition ...
Circle drawing

The screen origin (0,0) is at the top left.

\[ \text{point } a = (x - \sqrt{r^2 - \delta y^2}, y + \delta y) \quad \text{point } b = (x + \sqrt{r^2 - \delta y^2}, y + \delta y) \]

\[ \text{drawCircle}(x, y, r): \]

for each \( \delta y \in -r \ldots r \) do

\[ \text{drawLine from } (x - \sqrt{r^2 - \delta y^2}, y + \delta y) \text{ to } (x + \sqrt{r^2 - \delta y^2}, y + \delta y) \]
Character output primitives (in the Jack OS)

```java
class Output {
    function void moveCursor(int i, int j)
    function void printChar(char c)
    function void printString(String s)
    function void printInt(int i)
    function void println()
    function void backSpace()
}
```
Character output

- **Given:** a physical screen, say 256 rows by 512 columns
- We can allocate an 11 by 8 grid for each character
- Hence, our output package should manage a 23 lines by 64 characters screen
- Each displayable character must have a bitmap
- In addition, we have to manage a “cursor”. 
A font implementation (in the Jack OS)

class Output {
  static Array charMaps;
  function void initMap() {
    let charMaps = Array.new(127);
    // Assign a bitmap for each character
    do Output.create(32,0,0,0,0,0,0,0,0,0,0,0);          // space
    do Output.create(33,12,30,30,30,12,12,0,12,12,0,0);  // !
    do Output.create(34,54,54,20,0,0,0,0,0,0,0,0);       // ”
    do Output.create(35,0,18,18,63,18,18,63,18,18,0,0);  // #
    ... 
    do Output.create(48,12,30,51,51,51,51,51,30,12,0,0); // 0
    do Output.create(49,12,14,15,12,12,12,12,12,63,0,0); // 1
    do Output.create(50,30,51,48,24,12,6,3,51,63,0,0);   // 2
    ... 
    do Output.create(65,0,0,0,0,0,0,0,0,0,0,0);           // A ** TO BE FILLED **
    do Output.create(66,31,51,51,51,31,51,51,51,51,31,0,0); // B
    do Output.create(67,28,54,35,3,3,3,35,54,28,0,0);    // C
    ... 
    return;
  }
}

// Creates a character map array
function void create(int index, int a, int b, int c, int d, int e,
  int f, int g, int h, int i, int j, int k) {
  var Array map;
  let map = Array.new(11);
  let charMaps[index] = map;
  let map[0] = a;
  let map[1] = b;
  let map[2] = c;
  ...
  let map[10] = k;
  return; }

Keyboard primitives (in the Jack OS)

Class **Keyboard** {

    function char keyPressed()
    function char readChar()
    function String readLine(String message)
    function int readInt(String message)

}
Keyboard input

If the RAM address of the keyboard’s memory map is known, can be implemented using a peek function.

Problem I: the elapsed time between a “key press” and key release events is unpredictable.

Problem II: when pressing a key, the user should get some visible feedback (cursor, echo, ...).

**keyPressed()**:  
// Depends on the specifics of the keyboard interface  
if a key is presently pressed on the keyboard  
return the ASCII value of the key  
else  
return 0
Keyboard input (cont.)

**readChar()**:  
// Read and echo a single character  
display the cursor  
while no key is pressed on the keyboard  
    do nothing // wait till the user presses a key  
  c = code of currently pressed key  
while a key is pressed  
    do nothing // wait for the user to let go  
print c at the current cursor location  
move the cursor one position to the right  
return c

**readLine()**:  
// Read and echo a “line” (until newline)  
  s = empty string  
repeat  
  c = readChar()  
  if c = newline character  
    print newline  
    print newline  
    return s  
else if c = backspace character  
    remove last character from s  
    move the cursor 1 position back  
else  
    s = s.append(c)  
return s
Jack OS recap

- Implementation: similar to how GNU Unix and Linux were built:
- Start with an existing system, and gradually replace it with a new system, one library at a time.
Perspective

- What we presented can be described as a:
  - Mini OS
  - Standard library

- Many classical OS functions are missing

- No separation between user mode and OS mode

- Some algorithms (e.g. multiplication and division) are standard

- Other algorithms (e.g. line- and circle-drawing) can be accelerated with special hardware

- And, by the way, we’ve just finished building the computer.
The End