Operating Systems

Notice that, before you can run programs that you write in JavaScript, you need to jump through a few hoops first

JavaScript interpreter	CPU, machine language, bits
Web browser	
menu / icon / dock	
???	
login	
???	
turn on computer	

Many of those hoops involve the computer's *operating system*, or *OS* for short

- Let's backtrack a bit from your JavaScript program:

  ◇ Your JavaScript program was run from a Web browser

  ◇ You started your Web browser either from:

    - An onscreen menu of some sort

    - The browser's icon in some window

  ◇ The browser's menu item or icon is "just there" — readily available when you turn on or login to the computer that you were using

- The last bullet is where most users' experience ends; everything from there to the CPU seems either mysterious or magical

- It isn't magic…it's your operating system (OS)

What is an Operating System?

- First, some examples of operating systems:

  ◇ Microsoft Windows (XP, 2000, NT, ME, 98, 95…)

  ◇ Mac OS (X or otherwise)

  ◇ Linux (Debian, Red Hat, Ubuntu…)

  ◇ Unix (BSD, Solaris, System V…for that matter, Linux and Mac OS X are built on Unix foundations too; moreover, a lot of Unix software, including all of Linux and parts of Mac OS X, is *open source*, meaning that you can download, study, and modify the *source code* of these operating systems to fix bugs or add features)

  ◇ Palm OS

- And some others that don't always have names — iPods, cell phones, game consoles, and other devices have operating systems too

- An operating system is a program, just like everything else that runs on a computer; it is, however, a "very special" program — its job is to manage the overall computer on which it runs, so that all other programs can do useful work on this computer

- In many ways, the operating system is the intermediary between the programs we know — Web browsers, e-mail clients, word processors, games, music players — and the hardware on which they run: the CPU, main memory, their attached devices, the network, etc.

- Classically, an operating system handles four major tasks within a computer: *process management*, *memory management*, *storage*, and *input/output* (or "I/O")

- Not surprisingly, these tasks are interrelated

A Little History

- Once upon a time, there were no operating systems:

  ◇ Human beings prepared computers by hand in order to perform a task (punch cards, switches, teletypes)

  ◇ People "scheduled time" on computers the way we handle other scheduled activities — reservations on paper, maybe a sign-in sheet, maybe getting in line

  ◇ Computers just got a program to run, ran it (receiving input and providing output), then stopped; people would setup and clean up for every *job*

- Operating systems started as a mechanism for simplifying and automating this process

  ◇ Users wanting to run programs had to "submit" them in some specified form — the code itself, any input, and any other instructions like "save results to tape"

  ◇ The information was piled into a *job queue*; the computer reads one job at a time (first-come first-served), performs it, then goes to the next job

  ◇ This is *batch processing* — the jobs in the queue form a "batch of work" to be done in sequence

  ◇ The *control program* that cycled through reading then running each job is what evolved into today's operating systems

Programming Aside: Batch Processing in JavaScript

The following JavaScript program serves two purposes:

- It provides a miniscule example of how a batch processing control program — the precursor of modern operating systems — might have looked in programming language code

- It introduces a few new constructs that broaden the types of algorithms that you can make a computer perform — these constructs, while presented here in JavaScript, also have equivalent versions in most modern programming languages

```
function fivePlusFive() {
    return 5 + 5;
}

function zeroFarenheitToCelsius() {
    return -32 * 5 / 9;
}

function nickelsIn42Cents() {
    return parseInt(42 / 5);
}

// The "batch" to be processed, expressed as an array.
var jobs = [fivePlusFive, zeroFarenheitToCelsius, nickelsIn42Cents];

// The "batch processing" sequence.
var i = 0;
while (i < jobs.length) {
    alert(jobs[i]());
    i = i + 1;
}
```

Functions allow you to give names to individual subtasks

Arrays allow you to define lists of items; you can then manipulate the list as a whole (note "*jobs.length*" below) or access its members via a numerical *index* (e.g., "*jobs[i]*")

Loops let you perform something repeatedly — they usually include a condition to test that tells us when to stop repeating ourselves

The *jobs* array is a list of functions, so we can invoke them by adding parentheses to the end

The Need to Interact

- Batch processing was fine for tasks that can be left alone without further user intervention

- But as computers evolved, new applications that required an on-going dialog between the user and the computer — *interactive processing* — meant that users needed to stay "in front" of the computer to help direct or guide the task that it was performing

- However, users seldom had computers to themselves — their cost and size required that a single computer be *shared* by multiple interactive users concurrently

- The need for *time-sharing* of a single computer among multiple interactive users required a new leap in operating system capability

  ◇ Instead of just repeatedly getting one job, then running it, then going to the next one, the operating system needed to jump from one user to another in quick succession, so that each user felt as if he or she were the only one interacting with the machine

  ◇ This ability to *multitask* required that an operating system be able to suspend a program in mid-execution, transfer to another program, then pick up where it left off with the suspended program

- This operational model applies today even to a single user, as that user runs multiple programs concurrently

From Jobs to Everything Else

All other tasks in today's operating systems (memory management, storage, I/O, etc.) stem from the need to manage the resources required by jobs or programs — called *processes* today — in order to get their work done

- Processes need memory in which to run and work

- Many processes need to store the results of their work in a non-volatile form

- Processes, particularly interactive ones, need to receive data from the real world, and send data back

It Hasn't Stopped Yet

Just as room-sized computers and interactive applications motivated the birth and evolution of operating systems, new technologies and needs continue this evolution today

- Improving display technologies spawned *graphical user interfaces* and now motivate *convergence* with consumer electronics (HD, digital media)

- Multiprocessor machines result in the need to provide *parallelism*, *load balancing*, and *scalability*

- Communications technologies (Internet, wireless, etc.) amplify the need for better *security* and *privacy*

Types of Software

- While we have said that an operating system is "just another piece of software," it certainly does very different things from, say, a Web browser

- Because of the wide variety of tasks and roles taken by different programs, we tend to break software down into different categories

- In the end, however, by the time a program "reaches" the CPU as machine language, there is no longer a distinction (generally*) between which instructions came from the OS or from an end-user application

- It does still help to break software up into general, non-dogmatic categories:

  - ◇ *Applications*: Software that performs the tasks expected of a particular machine (preparing documents, playing games, displaying video, etc.)

  - ◇ *Utilities*: Software that helps maintain, monitor, and manage a machine; frequently interacts closely with the operating system

  - ◇ *Shell*: Software that helps the operating system communicate with the user

  - ◇ *Kernel*: The heart of an operating system, providing its most crucial and fundamental functions

* This is actually a bit of a white lie, but explaining it fully would require knowledge of kernel and user modes in modern CPUs, and…well, that's TMI for now

Operating System Components

- Depending on your perspective, software *utilities* and *shell* may or may not be part of an operating system; only the *kernel* tends to escape debate

- Operating system shells originally consisted of a *command line* interaction style: the user types individual commands which the computer then executes

  ◇ Today, most users expect a *graphical* shell, where input is expected not only from a keyboard but also a pointing device such as a mouse

  ◇ In most operating systems, more than one "style" of shell is available, in order to provide the user with the best environment for a particular task

- Within the kernel itself, a number of other distinct pieces of software provide specific functions

  ◇ *Device drivers* take care of details regarding how software should interact with hardware devices that are connected to the computer

    - Ideally, the set of available device drivers should form a *hardware abstraction layer* — a software representation of the computer that facilitates effective communication with connected devices without having to know the gory details of very specific device

  ◇ The *file manager* presents mass storage devices in the abstraction of a *file system*, typically consisting of files, directories or folders, and volumes

    - Not surprisingly, communication with said mass storage devices is taken care of through those devices' device drivers

  ◇ The *memory manager* takes care of allocating memory to running processes; due to the limitations of physical main memory, modern operating systems provide *virtual memory* that allow software to think that there is more main memory than there really is

  ◇ *Process management* components include the *scheduler* and the *dispatcher*: the scheduler decides when and for how long a process should be "serviced" by the CPU(s), while the dispatcher performs the actual dirty work of getting those processes to run

Booting

- As mentioned before, you start your JavaScript programs through a Web browser; in turn, you started your Web browser from some menu or window; this menu or window, in turn, probably started up when you logged into the computer…and so on

- Unlike the chicken-and-egg dilemma, this cycle of programs starting other programs does have an unequivocal origin — and this is called *booting*

- In the context of our previous discussion, booting tells us how the CPU gets its first instructions to execute

Despite their broad variety (PCs, game consoles, handhelds, music players, cell phones…), computers actually follow the same general *bootstrap sequence*:

- First, a CPU always reads its first instruction from some publicly known, designated address — information provided by the CPU's manufacturer

- Because this address must include valid instructions from the get-go, this very first *bootstrap program* is usually stored in *non-volatile* memory

- The bootstrap program's primary job is to locate, read, and run the rest of the computer's operating system; on PCs, this usually resides in mass storage, while on many other devices, this is some other type of non-volatile memory

Processes

- The usual conclusion of the bootstrap sequence is the execution of "process zero" — the very first "official" process in the operating system's life

- This first process then spawns everything else, all the way down to the Web browser that you open or even the JavaScript program that you write and run

- The operating system keeps track of the list of running processes in a *process table*; each entry in this table represents an individual process, and includes bookkeeping information regarding the process's *state*

Interrupts

- Having talked about an operating system's availability to switch across multiple programs, you might have wondered how the CPU knows to "get out" of a particular set of instructions in order to run another

- Modern systems accomplish this through a hardware event called an *interrupt*, which does what its name says: it interrupts what is going on in the CPU and forces it to jump to another section of main memory

- These *interrupt handlers* include process scheduling functions to interaction with hardware devices

Security

- An emerging function that is expected of operating systems is to ensure the *security* of the computer that the OS is managing

- Traditionally, an OS was expected only to "work correctly" — that is, protect against serious errors such as crashes, freezes, or data loss

- Multi-user operating systems introduced the concept of individual *users* running different programs — this expanded their role to that of protecting users from each other, either through files or programs

- In these historical contexts, the notion of an *administrator* or *superuser* evolved: operating systems necessarily had to provide a facility that granted total control over a system, for maintenance, upgrade, monitoring, and other purposes

- In today's environment, these constructs now have an added burden of ensuring that they are not compromised by malicious (or incompetent) parties:

  ◇ *Breach of confidentiality* is an attack that discovers information (such as passwords) that the attacker isn't normally allowed to see

  ◇ *Escalation of privileges* is an attack that inappropriately grants superuser "powers" to the attacker — usually made possible due to a successful breach of confidentiality

  ◇ *Trojan horses* are items that represent themselves as one thing but are really something else, such as enclosed "image" that is really a virus program — such entities may lead to further attacks, such as breach of confidentiality or escalation of privileges, or may cause harm within the attacked user's own account, such as deleting all of that user's files