Operating System

Operating systems are programs that provide a bridge between a computer's hardware and its application software. They have two primary purposes:

- managing resources; and
- resource abstraction.

Although this list is short, the operating system is usually the most complex and important piece of software that runs on our computers. Understanding *how* it works is a topic for another course, but it is important that we all know *what* it does and how we, as application writers and computer users, interact with the operating system to perform important tasks.

Managing Resources

The primary resources of a computer are

- its central processing unit (CPU),
- its memory hierarchy,
- its filesystem, and
- its peripherals.

CPU Managing the processor amounts to scheduling and coordinating processes. Processes are running programs. For example, when you launch a web browser, the operating system creates a process for that execution.

Memory The memory hierarchy is loosely partitioned into cache, main memory (e.g., random access memory) and secondary storage (e.g., hard disks). Memory higher in the hierarchy is usually faster and more expensive so the size of memory naturally increases as you descend the hierarchy. Cache and main memory are volatile storage devices (the contents are lost when powered down) while secondary storage is non-volatile. When a program is running it must be stored in main memory. Managing the memory often amounts to swapping data between memory units in the hierarchy.

Filesystem The filesystem is how an operating system organizes its data. Logical groups of data are stored in files, which are identified by a name. Files are organized in directories which are also named. Files have associated meta-data, which often includes information about permissions and size.

Peripherals The peripherals of a computer include the keyboard, mouse, monitor, network card, and printer. In the UNIX operating system, which we describe below, peripherals are treated as files.

Unfortunately, there are only so many resources available at once, and our operating system is forced to make some difficult decisions. We rely on our operating systems to divide the resources *fairly* among all of the processes running on our system, *secure* the resources according to a reasonable policy, and *isolate* problems in the (not uncommon) case of an error. A big part of our relationship with the operating system is *trust*. If we can't trust the OS, we are in trouble.

Resource Abstraction

An operating system's *kernel* is the part of the OS that runs with the most privilege. The kernel is primarily responsible for managing resources, but it also provides a well-defined interface so that applications can communicate with the hardware and resources of the computer. This interface is provided through a standard set of functions called *system calls*. Although every operating system defines its own unique set of system calls, each set performs roughly the same group of "core tasks". System calls let you do things like:

- create, delete, read, and write files
- request memory, and return it when you are done
- create a new process to run an application
- communicate with external devices like the network
- specify ownership and permissions of resources and objects

The system calls usually communicate directly with the kernel. In fact, your application tells the kernel what it wants to do, and then the kernel acts on their behalf. This distinction between an unpriviliged application and the privileged kernel helps to protect applications from carrying out potentially damaging actions.

UNIX

Unix is an operating system first developed by Ken Thompson and Dennis Ritchie at Bell Labs in the late 60's. It is the basis for OS X, the modern Macintosh operating system and the inspiration behind Linux, which is an operating system running many of the world's web servers. It is the operating system we will make exclusive use of in Diving into the Deluge of Data.

Shell

The shell is meant as the command line interface to the operating system. Among other things, it allows you to navigate and manipulate the filesystem and its contents (i.e., listing, creating, moving, removing, modifying, and changing the permissions files and directories) and run programs (i.e. create and manage processes).

The shell is a program that is usually executed at the start of a terminal session. To start a shell, you first need to open a terminal (the word terminal here is a throwback to actual terminal machines, which were used to connect to servers and mainframes). On OS X, this is done by launching the Terminal application located in `/Applications/Utilities`. By default the terminal will launch the shell associated with your login (imagine that the computer is a server and you're logging in remotely), which on OS X is Bash. To see your default login shell type

```
$ echo $SHELL
/bin/bash
```

Let's dissect this a bit. The `echo` command just echoes back its arguments to the terminal. Here's an example.

```
$ echo diving into the deluge of data
diving into the deluge of data
```

Bash maintains an environment, which has variables. You can see all the assigned variables by typing `env`. The `SHELL` environment variable stores the default login shell. To evaluate the contents of the variable requires prefacing the variable name with a dollar sign. Echoing this value back via `echo $SHELL` shows us the login shell. Finally, `/bin/bash` is the location that the Bash shell is stored in our file system.

Bash

Bash stands for *Bourne-again Shell*, which is a play on an older shell called *sh* created by Stephen Bourne. It is the default shell on all OS X and Gnu/Linux systems.

Version Control Systems

Almost everyone has a story of losing important data because they forgot to save. What is less obvious is that we all lose data whenever we *do* remember to save. Why is this?

When we save a file, we often *overwrite* one version of our file with a new one. The old version completely disappears. What happens if we didn't like our changes? What happens if we want to keep some of our new data, but restore something important from an older version? We are out of luck because we only have access to a single view of our data—the current view.

Version control systems were created to solve this problem. They preserve the intermediate states of our data as our files evolve. In other words, version control systems preserve file histories, which lets us track changes, roll backwards, and undo mistakes. Modern version control systems are also a useful way to share a project within a team and work seemlessly on different machines. We will be using `git`, a modern, distributed version control system in this course to do just that.

Programming Languages: Compiled verus Interpreted

Ultimately, every program that gets executed by the operating system is translated into assembly code and then into machine or object code. Assembly code is still readable by humans, but machine code is just a bunch of binary symbols—0s and 1s.

A *compiled* language means that the source code is translated directly to machine code via a compilation step (this translates the code from source to assembly language) followed by an assembly step (this translates the code from assembly directly into machine language).

An *interpreted* language means that the source code is not translated directly into machine language, but rather, evaluated by an interpreter. Because of this, interpreted languages often have two modes of operation: interactive and script. Interactive mode is often called the *read-evaluate-print-loop* or REPL because one interacts with the interpreter by writing some code that is then *read* and *evaluated*, the result of which is finally *printed*.

Here's an example in the programming language of the REPL:

```
>>> 27/3
9.0
>>> 1 + 1
2
```

Script mode just means that one records a full program in a file and then runs it through the interpreter in one fell swoop. To run `mycoolprogram.py` in script mode, you could open your terminal and type:

```
$ python mycoolprogram.py
```

And your system would execute the lines in `mycoolprogram.py`, one at a time, in order.