Week 3: Operating system

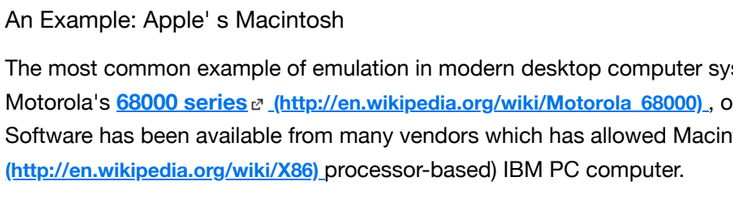
The operating system provides a layer between these hardware primitives and the end-user software applications, and provides a consistent method for applications to call upon the hardware.

They also take care of many housekeeping tasks, such as how the memory and disk resources are to be managed, or for scheduling which programs have control of the computer at any one time.

Application requirements

It is said that applications are written to a particular platform: for example, a "Windows program" requires Microsoft's Windows operating system to work.

- Provides a layer between hardware and user applications
  - attempts to protect hardware from user
  - manages resources in efficient and ‘fair’ manner
  - hides hardware details from user and application programmer



What this means is that the application has been written to interoperate with the application services that the particular operating system provides. Modern operating systems are often rich in what kinds of functionality they provide, which include (but are not limited to):

- **screen display:** instead of dealing with screen drawing directly, an application can request that "a window be drawn here, and some buttons there".
- **printing:** applications do not need to interface directly with printers; they merely describe how the page should look, leaving the actual task of communicating with the printer to a *printer driver* .
- **networking:** the operating system sets up and manages the network infrastructure, so that application software can use it easily.
- **file management :** the operating system provides a consistent interface to applications for dealing with memory and files, so data from different applications can co-exist.

Emulation

Earlier in the chapter, a big deal was made about a computer being Turing-complete; that is, one that can successfully perform any kind of logic operation. Based on this, it is technically possible for one Turing-complete machine to act as if it were another one. This process is called ***em u lat ion*** (http://en.wikipedia.org/wiki/Emulation) . This is a useful feature, as it allows computers to pretend that hardware exists.

For example, an application program might make some advanced graphics calls that require the use of a specialist graphics processor. If that processor isn't present in the system, it can emulate that processor and still fulfil the task.

Emulation does have a negative side-effect: it is very slow. It is not uncommon for an emulated environment to be many times slower than the 'real' version. This is because although two machines may be capable of similar tasks, there are underlying architectural differences which make working *exactly* alike quite cumbersome.

An Example: Apple' s Macintosh

The most common example of emulation in modern desktop computer systems has been on Apple's legacy Macintosh platform, which runs on Motorola's 68000 series (http://en.wikipedia.org/wiki/Motorola_68000) , or PowerPC (http://en.wikipedia.org/wiki/PowerPC) processors. Software has been available from many vendors which has allowed Macintosh computers to emulate an (Intel x86 (http://en.wikipedia.org/wiki/X86) processor-based) IBM PC computer.

The operating system itself also contains significant emulation; throughout the history of the Macintosh platform, there have been two changes to the underlying architecture which were fundamentally incompatible: 68000 to PowerPC, and then PowerPC to Intel x86.

The most recent of these changes occurred late in 2005 when Apple changed processor vendors (http://en.wikipedia.org/wiki/Apple%27s_transition_to_Intel_processors) from IBM (PowerPC) to Intel (x86)-- chips which are completely incompatible. To ensure that the transition from one architecture to the other was as smooth as possible, they developed an emulator called Rosetta (https://en.wikipedia.org/wiki/Rosetta_%28software%29) which allows software designed for the IBM chips to run on Intel processors.