CS162 Note 1: Intro

Kevin Moy

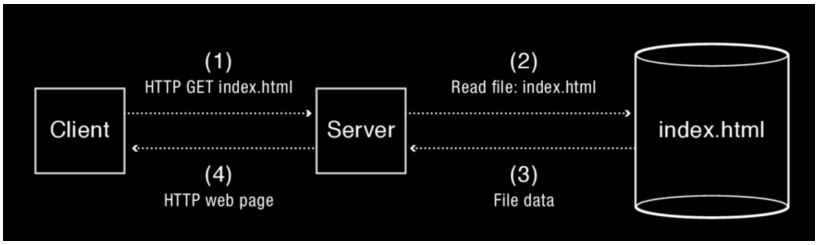
Abstract

A computer's operating system is by far its most important part. At a high level, it really handles the job of managing and allocating a computer's resources. Let's try to get a high-level overview of the importance, purpose, and functionality of the OS in this note, and specifically look at three main responsibilities of an operating system: manager, abstractionist, and glue.

1 Introduction

The operating system is present in literally every single electronic device you can think of- from the tiny chip in your car keys to the massive bundle of wires in a supercomputer. So really, we should be super familiar with operating systems- you're utilizing one to read this note right now! But most of that stuff has been abstracted away- in this course, we'll dig into a lot of the internals of the OS.

Let's consider the functionality of a simple web server to get started.



Pretty simple: the network is the client, and when it wants a webpage, it sends an HTTP GET request to our server. Server decodes the packet, reads the webpage contents from disk (or *cache*, ideally), and sends it back over

the network client which sends it back to the user machine, and we get our webpage.

One of the OS's jobs is to make it *easier* to write these server applications. But there are many other cases to handle, just for this case:

- **Communication**: for example, whenever user input happens for a webpage we need to run some code- specifically, *helper applications*, to make sure the correct *dynamic* webpage is returned. So the OS also needs to handle communication among processes and programs.
- **Multitasking**: How do we account for the (millions) of users who request a web page at the same instant? Certainly, handling shit one by one won't do it- if even one request gets stuck, everyone else has to wait. So we **multitask**: handle multiple requests at once- especially with multiprocessors.
- **Caching**: We definitely want to **cache** frequently requested web pages so we can return them to requests quicker. But now we have the issue of **shared data**, with all these requests for a single web page- so the OS needs to **synchronize** these accesses.
- **Security**: Web servers very often send javascript to make webpages fancier. But we need to prevent the chance that a virus overtakes the OS and compromises scripting code and allows it to send malicious scripts to clients!!

  ...and so on.

  Enough intro: let's delve into the OS.

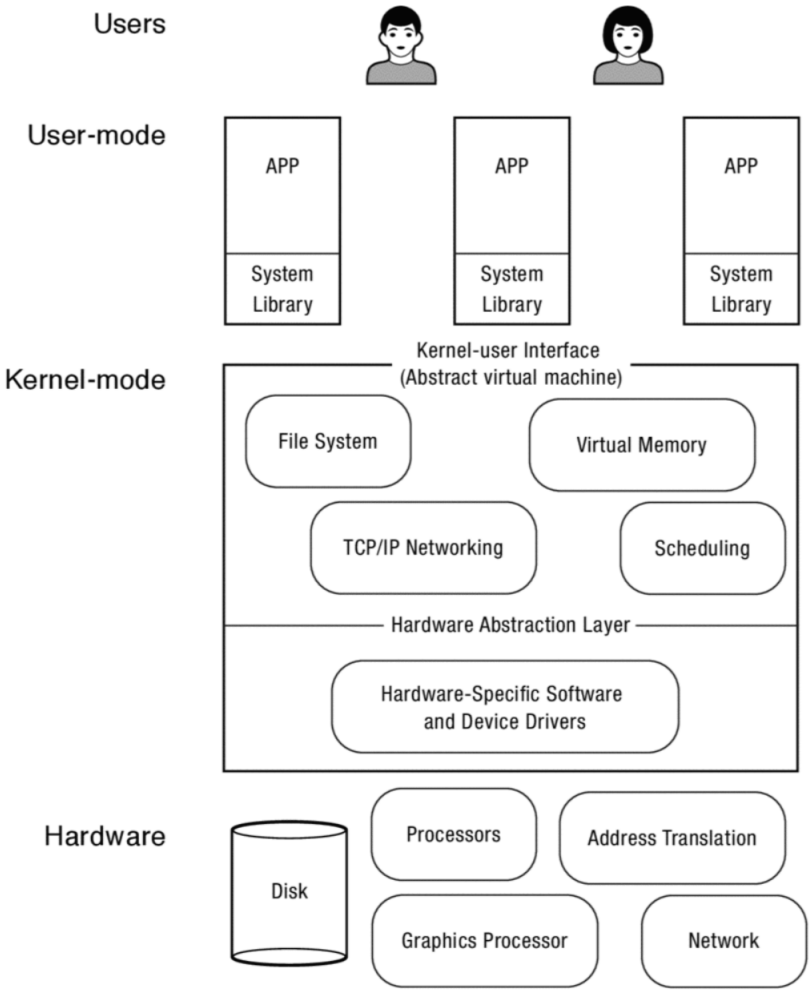
2 The Operating System

We really see OS's everywhere in electronics. In embedded systems, such as your XBOX or fridge, they are invisible for abstraction purposes. But they could also be plain as day and easily controllable in devices such as in smartphones and laptops. Pure ubiquity- as they should be.

The more a device depends on software, the more responsibilities it must actually take on: handling who can write car software, failure handling,

protecting from hackers, etc. One of the goals of this course is the importance of **reliability and security** in computer systems.

Generally, users interact with applications, which execute in an OS-made environment. The OS mediates access to the underlying hardware. Below is a diagram of a general-purpose OS:



The hardware is the lowest level of the computer: the processors, memory, and IO devices. The hardware also provides *primitives* for the OS so it can handle fault isolation and synchronization.

The OS, on the other hand, is the lowest level for *software*. There is a layer

for managing hardware and another layer for managing applications. The OS runs in an isolated execution environment completely separate from application code. A portion of the OS can also run as *system libraries* attached to each application.

Applications, then, run in this specially created **virtual environment**. Specifically, in order to run multiple applications safely and efficiently, the OS has to take three main responsibilities:

- **Referee**: The OS must share resources between different applications. It has the power to determine which programs can run at a given time. The OS also must *isolate* these applications such that a bug in one does not induce a bug in the other.
- **Illusionist**: The OS must abstract hardware from software design. Programmers should not need to worry about limitations of physical memory (unless they want to). Through virtual memory, the OS provides the *illusion* of nearly infinite memory. The OS resides at the abstraction layer *below* applications, so it is able to re-allocate memory as needed for each application, whenever.
- **Glue**: The OS must accomodate communication and sharing among applications. This is what allows copying and pasting from (most) different applications. Another role of the OS is to provide the UI such that applications have a sort of base similarity. Perhaps most importantly, the OS abstracts applications from IO devices, so applications can be written independently of these devices as well.

We will understand these three roles in much more detail as we look at the many examples of handling various application contexts.

3 Resource Sharing

The sharing of resources is *essential* for modern computers. Right now, you probably have more than one program pulled up on your computer. The OS must isolate these applications, yet also provide the illusion of full memory capacity to each. We also have to, of course, handle running multiple programs at the same time. Even single applications need to run multiple tasks at once: for example, a web browser rendering and loading a webpage

at the same time. On multiprocessors, we can exploit parallelization to make (parallel) programs run faster. The OS itself is an example of a parallel program!

So, with this feature, the OS once again finds itself with more responsibilities:

- **Resource Allocation**: A computers resources are finite: a set number of processors, memory, disk, and bandwidth. With multiple processes running at once, it is the OS's job to allocate these resources accordingly *so that we get the best overall performance*. Specifically, if a program becomes unresponsive, resource allocation ensures other programs can still run.
- **Fault Isolation**: Errors in a single program should not affect other programs *or even the OS*. If we can restrict a fuckup to its origin program and its origin program only, debugging will be far easier. This is a security thing too: something that happens in one program should certainly not open holes in another for a hacker. We also certainly don't want to let programs mess with the hardware either!
- **Communication**: Different applications/programs need to communicate with each other. So the OS must not only isolate bugs from other processes but also find a way to (safely) allow inter-process communication. It has to set these protocols and boundaries carefully.

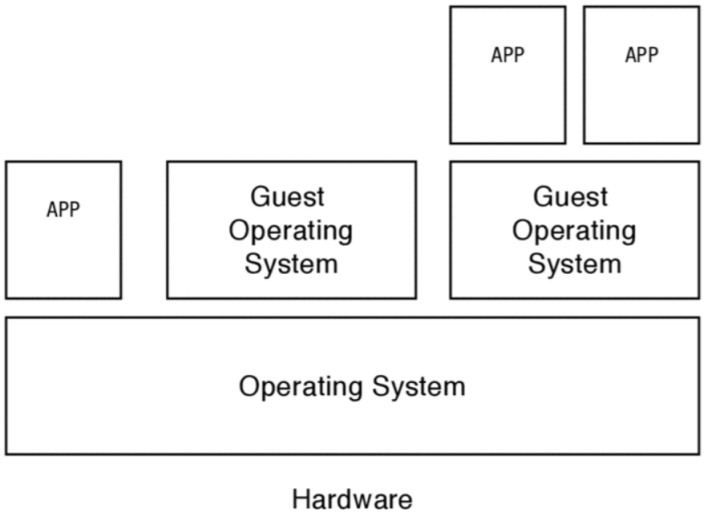
The OS balances needs, separates conflicts, and facilitates sharing. Enforcing and balancing these concerns is one of the key roles of the OS.

4 Masking Hardware Limitations

Physical hardware has obvious limitations- but the OS provides the illusion that this is not so. We know the OS has to allocate resources among different applications *dynamically*.

Virtualization is a process that gives a program the illusion that it has way more resources than it actually does. For example, the OS can make it seem like each application has its *own* processor, even though in reality only one processor exists. If wireless networks drop or corrupt packets, the OS covers up the fuckups.

We can even virtualize an entire computer by running another OS *as an application*! This is called creating a **virtual machine**:



The original OS provides the illusion that the **guest operating system** is running as its own OS on a real machine!

Why use virtual machines? Well, if you've ever tried to program an iOS app from Windows, you'd know a reason. Swift code only runs on IOS, but it'll run fine on a Hackintosh, since it *thinks* it's running on iOS. VMs are also good for debugging: we can debug an OS the same exact way (breakpoints, code-stepping) as normal programs, since a VM really is just another program masquerading as an OS!

5 Communication and Common Services

Let's end this note talking about our third OS role: providing **common services** to applications to abstract complexity and simplify design. For example, the OS handles the formatting of packets and the stupid shit from a web server, letting it allocate all its resources and "energy" to its main purpose (decoding and responding to GET requests). The OS provides the simple mechanism of data streams and reading/writing files.

The OS also provides common services is to allow applications to *communicate*. Web servers need to communicate with text editors to process their

requests. Any application that supports file sharing adheres to the standard file sharing/directory rules decided by the OS. Most operating systems also provide a (standard) way for applications to share memory.

The OS serves as the medium between applications and IO devices, providing a generic interface such that most (high-level) applications don't need to worry about what specific device with what capabilities is being used.

The OS also makes a computer look nice- the graphical user interface (GUI). Both Windows and Apple have some nice looking user-interface widgets. The OS tries to make this GUI "feel" as consistent as possible across different applications.

It is important to note that this third role of common services isn't that important and is actually a **byproduct** of the first two roles (resource sharing and hardware limitation masking). We'll focus much more on the first two roles for this course.