CS162 Note 3: Evaluating OS Quality

Kevin Moy

Abstract

The past two notes covered, at a *very* high level, the operating system's functionality as well as its responsibilities. We now move on to evaluating the design of an operating system, based its optimization of five standard characteristics: reliability/availability, security, portability, performance, and adoption.

1 Introduction

We know the OS is meant to handle a lot of shit. Because of the operating system's inherent complexity, we know that there are a lot of possible designs. How do we know if a design is better than others? Is there a "best" design? (Hint: No.) We'll cover this in this note.

2 Give Me Five

There are five main criteria for quality operating systems:

1. **Reliability+Availability**: The OS does what we want, when we want.
2. **Security**: The OS is resistant to hackers and bug-induced loopholes.
3. **Portability**: The OS can be used in different hardware platforms and can run legacy versions of application code.
4. **Performance**: The design of the OS is such that overhead is minimized, and the UI is fast.

5. **Adoption**: How many other users are there for this operating system?

Like pretty much everything, there is no "perfect" OS that maximizes all these traits. Tradeoffs are often inevitable. But these five things summarize what's most important in the design of an operating system.

Let's look at these criteria in more detail.

3 Reliability and Availability

Reliability means our OS does exactly what we want. This is the most important part of the OS and also the simplest: literally that it works correctly and doesn't fuck up. OS failures are usually far more serious than application failures- since application failure damage mitigation is literally one of the roles of the OS!

As simple as it sounds, making a reliable OS is still hard. Anyone can try sending malicious code to try and exploit the OS's vulnerabilities, so we have to make it rock-hard against these attacks. Unfortunately, implementing software reliability is a lot easier than OS (hardware) reliability: *everything* must work correctly for the OS to be reliable.

Availability means our OS is usable X amount of the time. We certainly don't want an OS that crashes every ten minutes. Note that a malicious user could have taken control of our computer but still made it functional for us (perhaps by installing a keystroke logger)- our OS is still available (but certainly not reliable OR secure).

Availability is affected by two factors:

1. Mean Time to Failure (MTTF)- a measure of failure frequency
2. Mean Time to Repair (MTTR)- a measure of restoration time

Ideally, then, to improve availability we want to increase the MTTF or reduce MTTR.

4 Security

Very closely related to reliability is **security**: the resistance of an operating system, or a computer in general, to malicious attackers.

No practical computer is perfectly secure. It'll be really hard to shield against EVERYTHING- because there will always be an attack vector available. Could be some exploitable bug in the system. Attackers might also find a way to tamper with hardware. Or even an internal issue with an untrustworthy administrator or developer can happen.

But we can try to minimize this vulnerability to attack. For example, strong fault isolation prevents third-party applications from creating holes in other programs. Downloading an app, no matter how shady, should not allow an attacker to install a virus (like a keylogger) on the computer.

Remember that applications themselves still have to account for security though. It is possible to forge an email with ANY sender email. This leads to all kinds of issues with virus emails. If the OS provided an ISOLATED attachment processing unit with limited capabilities, then we can severely limit the harm with email attachment viruses.

But the OS also has to allow access to sensitive or shared data in many cases. This becomes particularly important in communicating processes or programs: it's what allows copy-paste from Google to Microsoft Word. So we do still want to fault-isolate programs from each other, but STILL allow communication. We have to find a happy medium.

Thus, the modern OS needs an **enforcement mechanism** and a **security policy**. Enforcement is really just the OS, well, enforcing its rules and ensuring only permitted actions are executed. The security policy actually defines, in clear detail, WHAT is permitted. If either of these are compromised, it's clear that another attack vector is opened up!

5 Portability

A portable OS does not require application code to change as hardware changes. A Windows program should not depend on the graphics card, whether we have disk storage or flash, or whether we're using Bluetooth, WiFi, or Ethernet.

Portability also applies to the OS. The OS is magnificently complicated to create so we certainly don't want to remake it every time we have updates to hardware or software. New operating systems are *derived* from old ones. Operating systems normally last for decades.

So for operating systems to be portable, they must be designed to support (future) applications and be designed to run on (future) hardware. Code should not need to be changed with new advances in operating systems (and machines). It helps for applications to have a simple and standard way to interact with the OS: the **abstract virtual machine** (AVM). This interface includes:

- the API
- the memory access model
- Legal instructions (to execute)

Specifically regarding the third point, some instructions should be executable by the OS and the OS *only*.

A well-designed OS AVM is the medium that allows application code and hardware to evolve independently and still work! It's similar to how the Internet Protocol standard allows networking technology and distributed applications (email, Internet) to evolve as well. We should also note that changes in either hardware OR software will not require changes in the other.

The **hardware abstraction layer** (HAL) makes it such that the *operating system* can also be designed independent of hardware! What's the difference between HAL and the AVM? The AVM must do more. Remember we separate applications from hardware as much as possible- the OS is a lot closer to hardware so we don't want to abstract it *as much*.

Probably a universally accepted portable OS is Linux- used as an OS for pretty much every purpose imaginable. Based on the UNIX operating system, Linux is open-source and has grown to be compact, simple, highly portable, and very very popular.

6 Performance

Next up is performance- essentially, the speed of applications. The design of an operating system can greatly affect performance: the OS decides when applications can run, allocate memory to each one, cache vs. disk storage, etc. The OS really is the referee that mediates what applications can and cannot do- so it only makes sense it is a huge factor in performance.

Performance can be measured in multiple ways:

1. **Overhead**: Added cost of implementing some abstraction (for applications)
2. **Efficiency**: Lack of overhead in abstractions
3. **Response Time**: (Average) time for single tasks to run from start to finish
4. **Throughput**: task completion rate.

One thing to consider: would running the program directly on hardware, without the OS abstraction, improve performance by a lot? Would that cause more problems than it solves? If so, then the overhead does not outweigh the benefits of the abstraction.

Resource allocation can also affect performance. How do we divide resources among different applications? Do some get more or do all get the same? Which applications get priority if the former? These are all questions we'll tackle in later notes.

Another factor to consider is **performance predictability**: whether any performance metric is *consistent over time*. Simply taking the average won't mean shit if there's a lot of fluctuation. If a keystroke takes 10 seconds 1 percent of the time, this certainly won't be usable, even though the *average* response time is very good. Normal human beings would much prefer the response time be consistent over every scenario.

Let's consider the operating system as the road. Consider a dude driving. If the driver was the last person on earth, or was driving at 3 AM, there wouldn't be anyone else on the road, and he would zoom by. However, this is normally not the case- drivers need to share the road, inducing the need for control like stoplights and speed bumps. This leads to higher overhead

and response time for each driver- and predictability ultimately takes a hit. However, we can also increase throughput with carpooling or taking public transit.

7 Adoption

Finally, the success of an operating system depends on two more factors:

- Lots of apps that utilize the operating system
- Lots of hardware the OS can support

iOS is cool and all, but without all the fancy apps, the iPhone is just a Nokia 3120.

The **network effect** occurs when the value of technology depends not only on its material value but also on *popularity*. The more popular an OS is, the more applications and hardware that port to it. More users means more apps and cheaper hardware; more applications and cheaper hardware means more users.

So ideally we'd want to design our OS to make it easy to accommodate new hardware, as well as easy for applications to be ported across different versions of our OS.

A more subtle issue: should our API/OS source code be open-source or proprietary? Which is better? No right answer. It is pretty much universally known Windows/MacOS is proprietary and Linux is open-source; all three are very popular. Open systems are easier to adapt to different hardware platforms, but they risk **devolving into multiple versions**, which would limit the network effect. Some argue that proprietary systems are more reliable/better suited for customers. But if we limit an OS to one (hardware) platform, as companies like Apple do, we defeat the network effect and alienate customers (like me).

So we know being able to port applications from one OS to another helps an OS get established as standard- part of the network effect. However, if we make it such that it's DIFFICULT to port applications away from an OS, we can actually limit competition. How far do we go? That reaches into the business realm of things which we will not cover in this course.

8 Summary

In this note, we analyzed the five goals of the OS a little deeper and understood that in designing an operating system, tradeoffs are probably inevitable. Improving portability actually decreases reliability and security in general- since we have to account for more shit. If we break abstraction barriers in certain places, we could increase performance but add complexity and as a result (potentially) hurt reliability. So we just have to try the best we can as a designer!