CS 111

Warren Kim

July 14, 2023

Contents

1 Preface **2**

2 Overview **3**
- 2.1 What is an Operating System? . . . 3
- 2.2 Why Study Them? . . . 3
- 2.3 Key Topics (OS Wisdom) . . . 3
- 2.4 Why is the OS Special? . . . 4
- 2.5 Miscellaneous . . . 5
  - 2.5.1 Definitions . . . 5

3 Abstraction **5**
- 3.1 Why Abstract? . . . 6
  - 3.1.1 Corollary: Generalizing Abstractions . . . 6
- 3.2 Virtualization . . . 6
- 3.3 Resource Types . . . 6
  - 3.3.1 Serially Reusable . . . 6
  - 3.3.2 . . . 7
  - 3.3.3 . . . 7

1 Preface

Operating Systems: Three Easy Pieces

Text in these boxes will indicate that further details can be found in the textbook (*Operating Systems: Three Easy Pieces* by Arpaci-Dusseau)

Definitions

Any definitions will be appear in a grey box like this one. There may be more than one definition per box if the topics are dependent on each other or are closely related.

Examples

Any examples will be appear in a blue box like this one. Examples will typically showcase a scenario that emphasizes the importance of a particular topic.

2 Overview

This section defines what an operating system is as well as gives motivating reasons as to why we should be studying them.

2.1 What is an Operating System?

Definition: Operating System

An **operating system (OS)** is system software that acts as an intermediary between hardware and higher level applications (e.g. higher level system software, user processes), acting as an intermediary between the two. It manages hardware and software resources and provides common services for user programs.

The operating system plays a crucial role in managing hardware resources for programs, ensuring controlled sharing, privacy, and overseeing their execution. Moreover, it provides a layer of abstraction that enhances software portability.

2.2 Why Study Them?

We study operating systems because we rely on the ***services*** they offer.

Definition: Services

In the context of operating systems, **services** are functionality that is provided for by the operating system. They can be accessed via the operating system's API in the form of system calls.

Moreover, a lot of hard problems that we run into at the application layer have (probably) already been solved in the context of operating systems !

Example: Difficult Downloads

Suppose you are developing a web browser and implementing a *download* feature. While downloading things one by one works fine, what if you need to download multiple items from different sites simultaneously? Thinking abstractly, we can see that this is a problem of coordinating concurrent activities, and fortunately, this problem has already been solved in the context of operating systems! Since you have already learned how to tackle this issue in operating systems, now you can apply the same solution to your *download* problem!

2.3 Key Topics (OS Wisdom)

When thinking of how to solve complex problems, these are some things you should take into consideration (to hopefully make your life a lot easier).

Objects and Operations

Think of a service as an object with a set of well-defined operations. Moreover, thinking of the underlying data structure(s) of an object may be useful in many situations.

Interface v. Implementation

Definition: Interface and Implementation

An **interface** defines the collection of functionalities offered by your software. It specifies the method names, signatures, and the *purpose* of each component.

An **implementation** refers to the actual code that provides the functionality described by the interface. It specifies *how* the interface's operations are executed and realized in practice.

We separate the two components to improve modularity and create robust, well-structured code. It allows for different compliant implementations (as long as they adhere to the agreed-upon interface specifications). This provides immense flexibility at the implementation level!

Example: Sort Swapping

Assume you are writing a library that contains a collection of common algorithms, one of them being `sort()`. Being the genius that you are, your implementation is as follows: Randomly reorder the elements until they're sorted. By some miracle, your library garners a lot of attention, but users are complaining that `sort()` takes too long. Not knowing what's wrong with your implementation, you take DSA[a] and learn that you've got shit for brains. You want to rewrite `sort()` but are worried that it might break the interface. However, you remember that interface $\neq$ implementation, so you rewrite `sort()` (using something like merge sort) pushing this new implementation into production, and bragging about how `sort()` now runs in $O(n \log n)$ time.

[a] DSA: Data Structures and Algorithm

Encapsulation

We want to abstract away complexity (when appropriate) into an interface for ease of use.

Policy v. Mechanism

Definition: Policy and Mechanism

A **policy** is a high-level rule or guideline that governs the *behavior* of a system.

A **mechanism** is the implementation that is used to *enforce* the policy.

It is important to note that keeping policy and mechanism independent of one another is crucial. By separating policies from the underlying mechanisms, it becomes easier to change or modify policies without affecting the core functionality or technical implementation. This approach provides the ability to update policies independently from the underlying mechanisms, promoting modifiability, maintainability, and customization when designing software.

2.4 Why is the OS Special?

Definition: Standard and Privileged Instruction Set

The **standard instruction set** is the set of hardware instructions that can be executed by anybody.

The **privilaged instruction set** is the set of hardware instructions only the kernel can execute. When an application wants to execute a privilaged instruction, it must ask the kernel to execute it for them.

The OS is special for a number of reasons. Mainly, it has *complete* access to the privileged instruction set, *all* of memory and I/O, and mediates applications' access to hardware. This implies that the OS is *trusted* to always act in good faith. Thus, the OS stays up and running as long as the machine is still powered on (theoretically), and if the OS crashes, you're fucked lol.

2.5 Miscellaneous

Below are a list of miscellaneous topics.

2.5.1 Definitions

Definition: Instruction Set Architectures

An **instruction set architecture (ISA)** is the set of instructions supported by a computers. There are multiple (all incompatible) ISA's and they usually come in families.

ISA's usually come with privilaged and standard instruction sets.

Definition: Platform

A **platform** is the combination of hardware and software that provide an environment for running applications.

Common platforms include: Windows, [Mac, i]OS, Linux.

Definition: Binary Distribution Model

The **binary distribution model** is the paradigm of distributing software in compiled or machine code in the form of executables.

The binary distribution model is good for performance and security, but lacks in flexibility and is dependent on the platform you compile it for.

Definition: Binary Distribution Model

Portability refers to the ability to be adapted to different platforms with minimal modifications to the source code.

Portability is important if you're an OS designer because you want to maximize the number of people using your product $\implies$ your OS should run on many ISA's and make minimal assumptions about specific hardware.

3 Abstraction

Definition: Abstraction

Abstraction is the concept of providing a (relatively) simple interface to higher level programs, hiding unnecessary complexity.

The OS implements these abstract resources using physical resources, and thus is the source of one of the main dilemmas of OS design: What should you abstract?

Example: Network Neverland

Network cards consist of intricate technical details and specifications, but most users are not concerned with those details. As a result, the operating system abstracts the technical aspects of a specific network card, such as the process of sending a message, for higher-level programs. Instead of manually performing each step to send a message using a particular network card, users can simply call the OS's `send()`[a] function and let the operating system handle the complex operations.

[a] The actual function name may vary.

3.1 Why Abstract?

Abstraction is utilized to simplify code development and comprehension for programmers and users. Furthermore, it naturally fosters a highly modular codebase as each abstraction introduces an additional layer of modularity. Moreover, by concealing complexity at each layer of abstraction, it encourages programmers to concentrate on the essential functionality of a component.

3.1.1 Corollary: Generalizing Abstractions

Due to the variability of a machine's hardware and software, we can abstract the common functionality of each and make different types appear the same. This way, applications only need to interface with a common set of libraries.

Example: Printing Press

The portable document format (PDF) for printers abstracts away the individual implementation of a printer and provides a common format that all printers can recognize and print.

3.2 Virtualization

4) The Abstraction: The Process

Definition: Process

A **process** is an abstraction of a running program. It has an API

3.3 Resource Types

3.3.1 Serially Reusable

Definition: Serially Reusable Resource

Serially reusable resources are resources that can be used by a single process at a time (sequentially) and are not designed to be shared in parallel.

These resources require access control mechanisms to ensure that only one process can access them at any given time. This control ensures a graceful transition between users and prevents conflicts or data corruption that may arise from concurrent access.

Example: Printing Process

Printers are a serially reusable resource: multiple job can use it but only a single job will be printed at a time.

3.3.2

3.3.3