CS 111

Warren Kim

July 25, 2023

Contents

1 Preface **3**

2 Preface: Introduction to Abstraction **4**
2.1 Why Abstract? 4
2.1.1 Corollary: Generalizing Abstractions 4

3 Overview **4**
3.1 What is an Operating System? 4
3.2 Why Study Them? 5
3.3 Key Topics (OS Wisdom) 5
3.4 Why is the OS Special? 6
3.5 Miscellaneous 6
3.5.1 Definitions 7

4 Resource Types **7**
4.1 Serially Reusable 7
4.2 Partitionable 8
4.3 Shareable 8

5 Services **8**
5.1 Subroutines 9
5.2 Libraries 9
5.2.1 Bind Time 9
5.3 System Calls 10
5.3.1 Trusted Code 10
5.4 Messages 10
5.5 Middleware 11

6 Interfaces **11**
6.1 Application Programming Interface 11
6.2 Application Binary Interface 12
6.3 Corollary: Libraries 12
6.4 Best Practices for Interoperability 12
6.5 Aside: Side Effects 13

7 Abstraction **13**
7.1 Memory Abstractions 13
7.1.1 Complications 13
7.2 Interpreters 14
7.3 Communications 15
7.4 Generalizing Abstractions: Introduction to Federation Frameworks 16
7.5 Layering Abstractions 17

8 The Process **17**
8.1 Process Address Space 18
8.1.1 Layout 18
8.1.2 Code Segment 18
8.1.3 Data Segment 19
8.1.4 Stack Segment 19

1 Preface

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.

Abstractions

Layers of abstractions will appear in a violet box like this one. The key point of this box is to actively reinforce the concept of abstraction (and recognize how important it is in computer science!).

2 Preface: Introduction to 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.

2.1 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.

2.1.1 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 Overview

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

3.1 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.

Abstraction: The Operating System

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.

3.2 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!

3.3 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 Algorithms

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.

3.4 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.

3.5 Miscellaneous

Below are a list of miscellaneous topics.

3.5.1 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: Portability

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.

4 Resource Types

This section covers three types of OS resources: serially reusable, partitionable, and sharable.

Definition: Graceful Transition

A **graceful transitions** refers to the process of transferring control between two jobs such that there are no resource conflicts.

A graceful transition maintains system stability, data integrity and therefore cleanly releases resources. They typically ensure that users leave resources in a clean state; i.e. each subsequent user finds the resource in a "like new" condition.

4.1 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.

4.2 Partitionable

Definition: Partitionable Resource

Partitionable resources can be divided up (or *partitioned*) into smaller, disjoint[a] segments.

[a]Disjoint: Independent of one another.

These resources require access control mechanisms to ensure that each segment is *contained*[1] and *private*[2]. Partitionable resources can be temporarily allocated (e.g. RAM, CPU *time slice*, etc.) or permanently allocated (e.g. Disk storage[3], database tables, etc.).

Example: Memory Mania and Disk Division

Memory can be partitioned, allowing multiple unique processes to access their own allocated memory space independently.

Disk storage can be partitioned into separate logical volumes! This is commonly used to dual-boot different operating systems. Recently, M1(/2*ish*) Apple products can now run Linux on bare metal (still in beta !) via Asahi Linux!

Graceful transitions are still necessary in partitionable resources! Partitionable resources that aren't permanently allocated need to clean up after themselves.

4.3 Shareable

Definition: Shareable Resource

Shareable resources are usable by multiple *concurrent* clients. They need not "wait" for access nor do they "own" a particular subset of a given shared resource.

These resources require access control mechanisms to ensure that the shareable resource is used in a controlled and secured manner.

Example: Cloud Crazy

The cloud (e.g. Google Drive, Oracle Cloud, etc.) is a powerful shared resource! It enables multiple concurrent users to access shared folders and files, facilitating simultaneous editing and collaboration.

Graceful transitions typically are not necessary since a shareable resource generally doesn't change state *or* doesn't require any clean up. In the example above, while the cloud files change state, there is no cleaning up to do, so a graceful transition is not necessary (what's clean doesn't need cleaning).

5 Services

The OS provides services in a multitude of ways. This section will introduce and explain how services are provided throughout the software stack.

[1]Contained: Resources outside of a partition are not accessible.
[2]Private: External users cannot access the resources in your partition.
[3]Disk storage is permanently allocated until it isn't. You can use something like `fdisk` (in Linux) to modify partitions.

5.1 Subroutines

Definition: Subroutine

A **subroutine** in the context of operating systems is a small, self-contained and reusable portion of code that performs a specific function.

Subroutines are usually called directly, and operate like normal code (stack manipulation), and are typically seen in higher layers of the OS stack. They are fast, but usually cannot use the privileged instruction set. Furthermore, they are usually not language agnostic. The most common way to implement these subroutines are libraries!

5.2 Libraries

One way the OS provides services to users is via libraries. Standard utility functions such as `malloc` can be found in libraries (in this case, `stdlib.h`). So what exactly is a library?

Definition: Library

A **library** is a collection of code modules that encapsulate common operations, algorithms, and functionality.

Abstraction: Libraries

Most systems are equipped with a wide range of standard libraries, which are designed to be reused. These libraries encapsulate complexity and provide an additional layer of abstraction, simplifying problem-solving.

Example: DSA Doozy

In DSA, you likely had to implement different types of data structures (linear, hierarchical, graphical, etc.) and algorithms (search, divide/conquer, dynamic programming). Imagine your surprise when you find out most of these data structures and algorithms have already been written (and probably perform better than your implementation, no offense).

5.2.1 Bind Time

The choice of library bind time depends on multiple factors that include (but are not limited to): performance requirements, deployment and distribution considerations, and dependency management.

Definition: Static

Static libraries are precompiled code modules that link directly to the executable at compile time. They allow for efficient standalone executables, but they result in larger file sizes and require recompilation if the library is updated.

One easy example of a static library is `libc`, or the `C` standard library! It encapsulates a myriad of commonly used operations, algorithms, and functionality when programming in `C` like everyone's favorite `malloc()`.

Definition: Shared/Dynamic

Shared/Dynamically Linked libraries are separate files from the load modules that can be loaded and shared by multiple concurrent processes. They are loaded and linked to the executable during runtime.

One thing to note about shared libraries is that they cannot define or include global data storage. Moreover, called routines must be known at compile time since the fetching of the code is the only thing that is delayed until runtime.

5.3 System Calls

Definition: System Call

A **system call** is when users request functionalities from the operating system that are a part of the privileged instruction set.

System calls allow for the use of the privileged instruction set and interprocess communication, so why don't we just make everything a system call? System calls are *slow*[4]. Thus, we typically like to reserve system calls for operations that *require* the privileged instruction set.

Example: System Call Stress

System calls sound like a big deal, but you have already been introduced to them! Some include:

(i) File I/O: Reading from or writing to files on a disk require system calls since they require privileged instructions.

(ii) Memory management: Though functions like `malloc()` itself isn't a system call, it is implemented by calling system calls!

(iii) Process management: Only the kernel can *directly* create/destroy processes and ensure process privacy and containment.

(iv) Interprocess communication: Mainly for security reasons, communication between processes require privileged instructions.

Most of the time, everything related to a given system call is dealt with in the kernel. However, there are instances when the kernel will outsource tasks via direct calls to *untrusted*[5] code, waiting for a response, then returning to the calling process.

5.3.1 Trusted Code

Not all trusted code must be inside the kernel! If code doesn't *need* to access kernel data structures or execute privileged instructions, they might sit outside of the kernel layer.

Definition: Trust

Trusted code is guaranteed to be secure and will perform correctly. That is, it is safe for the operating system to run it.

Example: Lazy Login

A login manager/application is a great example of a program that is trusted but doesn't sit inside the kernel. When you login, the kernel will outsource the job to the login application!

5.4 Messages

Another way of service delivery is via messages that are exchanged with a server via system calls.

[4]System calls are between 100 and 1,000 times slower than standard subroutine calls!
[5]Untrusted: Not guaranteed to be secure.

Advantages and Disadvantages

Messages allow users to send and receive requests from anywhere! They are also highly scalable and can be implemeneted in user-mode code.	Messages are *slow*[a] and are limited to operate on process resources. [a]Messages are between 1,000 and 100,000 times slower than subroutine calls!

5.5 Middleware

Middleware refers to software components that are essential for a particular application or service platform but do not sit inside the OS. They bridge the gap between the application and underlying OS, providing additional functionalities and services.

Example: Middleware Madness

Some examples of middleware include database systems, web servers (like Apache and Nginx), distributed computing platforms (like Hadoop and Zookeeper), and cloud computing platforms like OpenStack.

We prefer middleware over implementing such functionalities directly in the kernel because kernel code is expensive and risky since kernel-level issues can impact the stability of the entire system! Instead, middlware is typically developed in user mode, making it easier to build, test, and debug[6]. Moreover, it is more portable, meaning it can be used across different operating systems without modification!

6 Interfaces

How do processes communicate with the OS? Interfaces! There are two main types: API and ABI.

Abstraction: Interfaces

Interfacess introduce a layer of abstraction, allowing programmers to focus on *what* they need without worrying about *how* it's implemented!

6.1 Application Programming Interface

Definition: Application Programming Interface

The **application programming interface (API)** provides a standardized set of rules and policies (at the source code level) that govern how different software components can communicate with each other.

API's are the basis for software portability, allowing a program to be compiled for a particular architecture or OS. That is, programmers can recompile for different targets without changing the source code, provided that the API is consistent. To do this, we link our program with OS-specific libraries that implement the funcitonality specified by the API.

Thus, when the program is compiled and linked using an API-compliant system, the binary executable will be compatible[7] with any other API-compliant system! Well-defined API's allow us to create interoperable applications and libraries that are platform/system independent, promoting software portability, reusability, and simplicity when building complex systems!

[6]If the middleware crashes, it can be restarted independently of the entire system, minimizing its impact on the overall OS stability.

[7]An API-compliant program will compile and run on any system that supports the same API.

6.2 Application Binary Interface

Definition: Application Binary Interface

The **application binary interface (ABI)** defines the low-level binary interface that allows compiled programs to interact with the underlying system and hardware.

ABI's govern how DLL's are structured and how they interact with programs at a binary level. But how? ABI's define how data is represented in memory and passed between functions and across different modules! This includes details like register usage, linkage conventions, parameter passing conventions, and stack layout. They also connect the API to the specific hardware, translating source-level instructions to actual machine level instructions.

Once a program is compiled using an ABI-compliant system, it can run unmodified (i.e. without recompilation) on any other system with the same ABI! This ensure that a single binary can service all ABI-compliant systems. Hence, they are usually intended for end-users and software deployment.

6.3 Corollary: Libraries

Libraries are accessed through API's! The API provides source-level definitions for how to access the library, and is readily portable between systems. While DLL's are also accessed via an API, the loading mechanism is specified by the ABI.

6.4 Best Practices for Interoperability

Standalone programs are useless! All useful programs use system calls, library routines, operate on external files, exchange messages, etc. That is, they all utilize OS services! Thus, if the interface changes, these programs will fail. Thus, API requirements are frozen (finalized) at compile time. This means:

(i) Execution platforms must support the interfaces.

(ii) All partners/services must support the interface protocols.

(iii) The API must be backwards compatible

That is, API's need to be *stable* in order to support interoperability! API's need to be rigorously specified and standardized[8]. Thus, the developers that use the API are encouraged to be compliant with the API specifications to ensure that their program will survive updates.

Example: New Version New Problems

Suppose you are writing an application for an OS running on version 0.6.8, and being the genius programmer you are, find an exploit that goes around the OS API to make your app run faster. Everything runs fine until version 0.6.9. Confused, you find that the developers hate you in particular and decided to patch the exploit you used. Having learned your lesson, you have to rewrite that entire feature, being compliant with the API.

Interoperability requires both parties to honor their side of the contract: Standard bodies must keep the interface stable, while developers must be compliant with the API if they want their products to survive new updates.

[8]Standard body: Most big projects (like Linux) have standard bodies that manage the interface definitions so as to maintain stability!

6.5 Aside: Side Effects

Definition: Side Effect

A **side effect** is occurs when an action on one object has *non-trivial*[a] consequences.

[a]Non-trivial: Effects that are not included in the interface specifications.

Side effects are inevitable, but are not the end of the world! They usually happen due to shared state between independent modules and functions. Thus, developers should ignore side effects and continue being compliant with the interface as they *should* be patched. In general, try to avoid exploiting side effects since they are not guaranteed to be there or maintained!

7 Abstraction

Recall that we like abstractions in Computer Science. Life is easy for high level programmers if they work with a simple abstraction. The OS is responsible for creating managing, and exporting such abstractions.

Example: Hardware Hiding

Hardware is fast, but complex and limited, so using it *correctly* can be extremely challenging. Thus, hardware is commonly seen as a building block rather than a solution. We provide abstractions to encapsulate implementation details (like error handling and performance optimization) and eliminate behavior that is irrelevant to the user. Thus, abstractions make it more convenient to work with the hardware!

The OS provides some core abstractions that our computational model relies on: memory, processor, and communication abstractions.

7.1 Memory Abstractions

Memory abstractions provide a consistent way to interact with various data storage resources, simplifying the process for users. However, there are some complicating factors that come with abstracting memory.

7.1.1 Complications

The operating system managing these abstractions doesn't have abstract devices having arbitrary properties. Instead, it handles physical devices that may have inconvenient properties. Therefore, the primary goal of OS abstraction is to create an abstract device with desirable properties derived from the physical device that lacks them.

Memory Lifetime

Definition: Persistent and Transient Memory

Persistent memory retains data even when power is turned off (long term memory). Examples of non-volatile storage include hard drives and solid state drives.

Transient memory loses its data when power is turned off (short term memory). An example of volatile storage is RAM.

Managing data in both types of memory presents different challenges and considerations when designing a complex system.

Example: File Finding

When you run a program, all of the variables local to the program are stored in transient memory, or RAM. However, suppose you write to a file `foo` in your program. `foo` is stored in persistent memory! This means that weeks later, if another program wants to access foo (e.g. `cat foo`), the contents you wrote into `foo` will still be there.

Size

There can be a discrepancy between the size of memory operations that the user wants to work with and the size that the physical memory can handle. Thus, being able to manage data manipulation efficiently, especially when data sizes differ, is very important!

Example: Caught in 4k

Hardware usually processes data at the word level[a]. However, when writing a block of data to flash memory, we typically move data in 4k chunks. Therefore, when reading from or writing to memory, we must ensure that data is moved in the appropriate word size.

[a]Word sizes are typically 32 or 64 bits depending on the architecture

Latency

Definition: Persistent and Transient Memory

Latency (in the context of memory) refers to the time it takes for a process to read from memory.

Note that the latency reading from RAM and from disk are two very different times which lead to varying performance gains depending on which one you optimize for.

Implementation Variety

The same memory abstraction might be implemented using various physical devices. This leads to varying performance depending on which device imlements the abstraction.

Example: Caught in 4k

Storage devices like hard disks, solid-state drives, and optical drives can all be used to implement file storage, but the performance differences vary between the three. (SSD > HD > optical).

7.2 Interpreters

Definition: Persistent and Transient Memory

An **interpreter** is the module (abstract or physical) that executes commands and "gets things done".

Abstraction: Interpreters

At the physical level, we have the CPU. Directly working with the CPU is not easy, so the OS provides a higher level intepreter!

An interpreter has several basic components:

(i) Instruction Reference: Tells the interpreter which instruction to execute next.

(ii) Repertoire: The set of features that the interpreter supports.

(iii) Environment Reference: Describes the current state on which the next instruction should be executed.

(iv) Interrupts: Situations in which the instruction reference pointer is overridden.

Example: Processing Processes

The process that we interface with is an example of an interpreter. The OS maintains the *instruction reference* (a program counter for a given process). Its source code specifies its *repertoire*, and its stack, heap, and register contents are its *environment*. The OS manages all three of these components. Another thing to note is that no other interpreters should be able to access the process' resources; i.e. the interpreter should be private.

Aside: Implementation

Implementing the process abstraction in the OS is relatively straightforward when dealing with only one process, but in reality, that is seldom the case. When dealing with multiple processes, we have to consider that:

(i) The OS has limited physical memory to hold environment information.

(ii) There are usually only one set of registers (or one per core).

(iii) The process shares the CPU (or core) with other processes!

To address these issues, we need:

(i) A *scheduler* to share the CPU among multiple processes.

(ii) Better *memory management* hardware and software to create the illusion that each process has full access to RAM (when in reality, they don't).

(iii) Access control mechanisms for other memory abstractions to keep our machine secure.

7.3 Communications

Definition: Communication Link

A **communication link** allows interpreters to talk to each other (on the same or different machines).

Abstraction: Communication Links

A communication link at the physical level consists of memory and cables. However, at more abstract levels, we have networks and interprocess communication mechanisms.

Communication links are distinct from memory abstractions for a couple of reasons:

(i) Factors such as network congestion, distance, and hardware limits contribute to variations in speed, latency, and bandwidth. On the other hand, memory access offers more predictability and consistent performance.

(ii) Communication links are often asynchronous[9], introducing additional complexity compared to synchronous memory access.

(iii) The receiver in communication links may be reactive, meaning they only perform an operation because the sender initiated it. In contrast, data is usually immediately available when requested via memory access.

[9]Asynchronous: Data can be sent and received independently of each other. Timing of communication is not guaranteed to be coordinated.

Aside: Implementation

If both ends of the communication link are on the same machine, it's trivial: use memory for transferring data! Copy the message from the sender's memory into the receiver's memory *or* transfer *control*[10] of the memory containing the message from the sender to the receiver. To implement communication links across machines, we have to consider the following:

(i) We need to optimize the cost of copying data.

(ii) Memory management can become very tricky (especially when manipulating ownership!).

(iii) We need to include complex network protocols into the OS itself. This raises new security concerns that the OS might need to address.

(iv) We need to be able to deal with message loss, retransmission, etc.

7.4 Generalizing Abstractions: Introduction to Federation Frameworks

Rather than applications dealing with varied resources, we can make many different things *appear*[11] the same by using a unifying model! Usually, these unifying models involve a federation framework.

Example: Computer Communism

A Portable Document Format, or PDF, is the unifying model for printed output. If we want to print something to a printer, as long as the document is in the PDF format, it will know how to print it!

SCSI, SATA, and SAS are standard ways to interface with hard disks and other storage devices (CD, SSD, etc.).

Definition: Federation Framework

A **federation framework** is a structural design that enables similar (but different) entities to be treated uniformly by creating a single *interface*[a] that all entities must adhere to.

[a]The *implementation* that supports the interface is specified by the particular entities.

Note that a unifying model need not be the model with the "lowest common denominator"[12]. Rather, the model can include "optional features" which are implemented in a standard way. Why? Some devices may have features that others of the same class do not. Thus, these "optional features" allow us to create a highly modular federation framework while maintaining a common unifying model.

Example: Pretty Printing

Suppose you are building a federation framework for printers. Some printers can only print single-sided while others can print double-sided. So, a possible federation framework could require that all devices that classify as printers *must* be able to print *at least* single-sided, with the *optional feature* of double-sided printing. Extending this idea, we can add more optional features like color printing, DPI settings, etc.

Unfortunately, there may be instances where a particular device may have features that cannot be exploited through a common model. This is the tradeoff we make for a uniform model. There have been arguments both for and against being able to handle such features in a federation framework.

[10]Transferring control: Change who owns the memory segment!
[11]We want to *abstract* away the implementation details!
[12]Lowest common denominator: The set of features *all* entities have in common.

7.5 Layering Abstractions

It is very common practice to create increasingly complex services by *layering* abstractions.

Example: Abstract Abstract Abstract!

A generic file system is an abstraction layer over a particular file system (1). A particular file system layers on top of an abstract disk (2). This abstract disk layers on a real disk (3). Here, a generic file system is implemented with 3 layers of abstraction! This hierarchical structure simplifies development and enhances system scalability.

Layering allows for modularity, easy development of multiple services on multiple layers, and flexibility in supporting various underlying services. Abstractions hide complex implementation details, promoting structured and independent design.

Unfortunately, layers in a system often introduce performance penalties due to the additional indirection they bring. Moving between layers can be costly as it typically involves changing data structures and representations, and may require extra instructions. Moreover, layers may not be entirely independent of one another; for example, lower layers can impose limitations on what upper layers can achieve.

Example: Packages Play Hide n Seek

An abstract network link may hide causes of packet loss, since the lower layer that this abstraction is built off of may hide certain implementation details that are relevant to these issues.

The OS offers numerous abstractions, catering to diverse needs and use cases. Selecting the most suitable abstractions is crucial for achieving optimal results. It involves understanding the trade-offs between higher-level and lower-level abstractions to ensure efficient utilization of system resources and effective application development.

8 The Process

Process and State

A **process** is a type of interpreter that executes an instance of a *program*[a].

[a]A **program** is a set of instructions that defines a particular application.

A **state** is a mode or condition of being, representable by a set of bits.

When you begin executing a program, it becomes a process. There may be multiple instances of the same program running simultaneously on the same computer. Typically in these cases, each running instance is a separate process.

Abstraction: Processes

Processes are a type of interpreter, an abstraction we covered in the previous section. We can think of it as a virtual private computer.

A process is an *object*[13], characterized by its state and its operations. All persistent objects have state, distinguishing them from other objects and characterizing the object's current condition. OS's objects' state is mostly managed by the OS itself and not by the user code. Thus, we must ask the OS to access or alter the state of an OS object.

[13]Object: NOT the OOP object.

Example: Priority Process

The OS maintains information about the current priority of each process in the system. This subset of *state* determines its position in the scheduling queue.

8.1 Process Address Space

Definition: Process Address Space

The **process address space** is the set of addresses visible to the process. This address space is *private*[a] to the process.

[a]Private: Inaccessible by outside processes.

The process' address space consists of all memory locations accessible by the process. Invalid addresses are those outside its address space, and as such, the process cannot request access to them. Modern operating systems give the illusion that *every* process' address space can (but often don't) include *all* of memory.

8.1.1 Layout

The process address space typically consists of different segments:

(i) Shared Code: Contains the executable code of the program. We do not do self-modifying code in modern computer systems. Thus, this shared code is static while the process is running, meaning they are read/executable only. So, subsequent instances of a program will be accessing the one stored in RAM since it's a *shared resource*.

(ii) Shared Libraries: Like shared code, shared libraries are a *shared resource* that are stored somewhere in RAM. Thus, multiple processes can access these shared libraries concurrently.

(iii) Private Data: Stores global and static variables used by the program. Since private data is read/write, they are *not* a shared resource, and thus is private to a particular instance of a process.

(iv) Private Stack: Like private data, the private stack is private to a particular instance of a process.

All of these must sit somewhere in RAM, but different type of memory elements have different requirements (e.g. *shared* coded is read/execute, *private* stack is read/write).

Linux Layout

Each operating system puts these process memory segments in different places, but here's how Linux does it! Code segments are statically sized and are put at the beginning (e.g. 0x00000000). The data segment (and heap) is placed after the code segment and grows upward. The stack is placed at the very end (e.g. 0xFFFFFFFF) and grows downward. It is crucial that the data segment and stack are **not** allowed to meet!

8.1.2 Code Segment

Load Module

The **load module** is the output of a linkage editor, where all external references have been resolved and object modules have been combined into a single executable.

The process starts by creating a load module. To make instructions executable, we need to load the code into RAM, as we can't directly run instructions from the disk. Next, we read the code from the load module and copy it into a specific *code segment*[14] within the process's address space. This allows the CPU to fetch and execute instructions from the memory while the program runs. Since the code is static (read/execute only), we can share it among multiple processes, which helps reduce unnecessary duplication of code.

8.1.3 Data Segment

Data also needs to be initialized within the address space of a process. This requires the creation and mapping of a process data segment into the process' address space. The initial contents of the data segment are copied from the load module. In particular, the BSS[15] segments are initialized to all zeroes. Data segments are read/write (and thus private to the instance of the process). The program can grow or shrink this segment (via the `sbrk` syscall).

8.1.4 Stack Segment

Stack Frame

A **stack frame** serves as storage for procedure local variables, invocation parameters, and save/restore registers.

Modern programming languages are stack-based. Thus, each procedure call allocates a new stack frame, and once it is completed, the corresponding stack frame is popped off of the stack, freeing any memory that was allocated for it. Modern CPU's have built-in stack support. Thus, the stack must be preserved as part of the process state to ensure proper execution and continuity during a process' lifetime.

The size of the stack in a program depends on its activities, such as the amount of local storage used by each routine. It grows larger as calls nest more deeply (since each procedure call allocates a new stack frame!), and once these calls return, their stack frames can be recycled for future use.

Who Manages The Stack?

The operating system is responsible for managing the process' stack segment! It is created alongside the data segment when the program is loaded into memory.

Different operating systems implement stack management differently Some allocate a fixed-size stack at the program's load time, while others dynamically extend the stack as the program needs more space.

Across all operating systems, stack segments are usually only read/write for security reasons. This prevents any unintended execution of code in the stack (e.g. buffer overflows) and ensures that it is used exclusively for storing data and variables.

Stack segments are process private, meaning each process has its own unique stack. This isolation ensures that processes cannot interfere with each other's stacks, which is crucial for maintaining system stability and security.

[14]Code segment: A segment we establish in the process' address space to accommodate the code.
[15]BSS: Block Started by Symbol