OS Interfaces and Abstractions
CS 111
Winter 2023
Operating System Principles
Peter Reiher

OS Interfaces

- Nobody buys a computer to run the OS
- The OS is meant to support other programs
  - Via its abstract services
- Usually intended to be very general
  - Supporting many different programs
- Interfaces are required between the OS and other programs to offer general services

Interfaces: APIs

- Application Program Interfaces
  - A source level interface, specifying:
    - Include files, data types, constants
    - Macros, routines and their parameters
- A basis for software portability
  - Recompile program for the desired architecture
  - Linkage edit with OS-specific libraries
  - Resulting binary runs on that architecture and OS
- An API compliant program will compile & run on any compliant system
  - APIs are primarily for programmers

APIs help you write programs for your OS

Interfaces: ABIs

- Application Binary Interfaces
  - A binary interface, specifying:
    - Dynamically loadable libraries (DLLs)
    - Data formats, calling sequences, linkage conventions
  - The binding of an API to a hardware architecture
- A basis for binary compatibility
  - One binary serves all customers for that hardware
    - E.g. all x86 Linux/BSD/MacOS/Solaris/…
- An ABI compliant program will run (unmodified) on any compliant system
- ABIs are primarily for users

ABIs help you <u>install</u> binaries on your OS

Libraries and Interfaces

- Normal libraries (shared and otherwise) are accessed through an API
  - Source-level definitions of how to access the library
  - Readily portable between different machines
- Dynamically loadable libraries also called through an API
  - But the dynamic loading mechanism is ABI-specific
  - Issues of word length, stack format, linkages, etc.

Interfaces and Interoperability

- Strong, stable interfaces are key to allowing programs to operate together
- Also key to allowing OS evolution
- You don’t want an OS upgrade to break your existing programs
- Which means the interface between the OS and those programs better not change

Interoperability Requires Stability

- No program is an island
  - Programs use system calls
  - Programs call library routines
  - Programs operate on external files
  - Programs exchange messages with other software
  - If interfaces change, programs fail
- API requirements are frozen at compile time
  - Execution platform must support those interfaces
  - All partners/services must support those protocols
  - All future upgrades must support older interfaces

Interoperability Requires Compliance

- Complete interoperability testing is impossible
  - Cannot test all applications on all platforms
  - Cannot test interoperability of all implementations
  - New apps and platforms are added continuously
- Instead, we focus on the interfaces
  - Interfaces are completely and rigorously specified
  - Standards bodies manage the interface definitions
  - Compliance suites validate the implementations
- And hope that sampled testing will suffice

Side Effects

- A *side effect* occurs when an action on one object has non-obvious consequences
  - Effects not specified by interfaces
  - Perhaps even to other objects
- Often due to shared state between seemingly independent modules and functions
- Side effects lead to unexpected behaviors
- And the resulting bugs can be hard to find
- In other words, not good

Tip: Avoid ***all*** side effects in complex systems!

Abstractions

- Many things an operating system handles are complex
  - Often due to varieties of hardware, software, configurations
- Life is easy for application programmers and users if they work with a simple abstraction
- The operating system creates, manages, and exports such abstractions

Simplifying Abstractions

- Hardware is fast, but complex and limited
  - Using it correctly is extremely complicated
  - It may not support the desired functionality
  - It is not a solution, but merely a building block
- Abstractions . . .
  - Encapsulate implementation details
    - Error handling, performance optimization
    - Eliminate behavior that is irrelevant to the user
  - Provide more convenient or powerful behavior
    - Operations better suited to user needs

Critical OS Abstractions

- The OS provides some core abstractions that our computational model relies on
  - And builds others on top of those
- Memory abstractions
- Processor abstractions
- Communications abstractions

Abstractions of Memory

- Many resources used by programs and people relate to data storage
  - Variables
  - Chunks of allocated memory
  - Files
  - Database records
  - Messages to be sent and received
- These all have some similar properties
  - You read them and you write them
  - But there are complications

Some Complicating Factors

- Persistent vs. transient memory
- Size of memory operations
  - Size the user/application wants to work with
  - Size the physical device actually works with
- Coherence and atomicity
- Latency
- Same abstraction might be implemented with many different physical devices
  - Possibly of very different types

Where Do the Complications Come From?

- At the bottom, the OS doesn't have abstract devices with arbitrary properties
- It has particular physical devices
  - With unchangeable, often inconvenient, properties
- The core OS abstraction problem:
  - Creating the abstract device with the desirable properties from the physical device that lacks them

An Example

- A typical file
- We can read or write the file
  - We can read or write arbitrary amounts of data
- If we write the file, we expect our next read to reflect the results of the write
  - *Coherence*
- We expect the entire read/write to occur
  - *Atomicity*
- If there are several reads/writes to the file, we expect them to occur in some order

What Is Implementing the File?

- Often a flash drive
- Flash drives have peculiar characteristics
  - Write-once (sort of) semantics
    - Re-writing requires an erase cycle
    - Which erases a whole block
    - And is slow
  - Atomicity of writing typically at word level
  - Blocks can only be erased so many times
- So the operating system needs to smooth out these oddities

What Does That Lead To?

- Different structures for the file system
  - Since you can’t easily overwrite data words in place
- Garbage collection to deal with blocks largely filled with inactive data
- Maintaining a pool of empty blocks
- Wear-leveling in use of blocks
- Something to provide desired atomicity of multi-word writes

Abstractions of Interpreters

- An interpreter is something that performs commands
- Basically, the element of a computer (abstract or physical) that gets things done
- At the physical level, we have a processor
- That level is not easy to use
- The OS provides us with higher level interpreter abstractions

Basic Interpreter Components

- An instruction reference
  - Tells the interpreter which instruction to do next
- A repertoire
  - The set of things the interpreter can do
- An environment reference
  - Describes the current state on which the next instruction should be performed
- Interrupts
  - Situations in which the instruction reference pointer is overridden

An Example

- A process
- The OS maintains a program counter for the process
  - An instruction reference
- Its source code specifies its repertoire
- Its stack, heap, and register contents are its environment
  - With the OS maintaining pointers to all of them
- No other interpreters should be able to mess up the process' resources

Implementing the Process Abstraction in the OS

- Easy if there's only one process
- But there are almost always multiple processes
- The OS has limited physical memory
  - To hold the environment information
- There is usually only one set of registers
  - Or one per core
- The process shares the CPU or core
  - With other processes

What Does That Lead To?

- Schedulers to share the CPU among various processes
- Memory management hardware and software
  - To multiplex memory use among the processes
  - Giving each the illusion of full exclusive use of memory
- Access control mechanisms for other memory abstractions
  - So other processes can’t fiddle with my files

Abstractions of Communications

- A communication link allows one interpreter to talk to another
  - On the same or different machines
- At the physical level, memory and cables
- At more abstract levels, networks and interprocess communication mechanisms
- Some similarities to memory abstractions
  - But also differences

Why Are Communication Links Distinct From Memory?

- Highly variable performance
- Often asynchronous
  - And usually issues with synchronizing the parties
- Receiver may only perform the operation because the send occurred
  - Unlike a typical read
- Additional complications when working with a remote machine

Implementing the Communications Link Abstraction in the OS

- Easy if both ends are on the same machine
  - Not so easy if they aren't
- On same machine, use memory for transfer
  - Copy message from sender's memory to receiver's
  - Or transfer control of memory containing the message from sender to receiver
- Again, more complicated when remote

What Does That Lead To?

- Need to optimize costs of copying
- Tricky memory management
- Inclusion of complex network protocols in the OS itself
- Worries about message loss, retransmission, etc.
- New security concerns that OS might need to address

Generalizing Abstractions

- How can applications deal with many varied resources?
- Make many different things appear the same
  – Applications can all deal with a single class
  – Often Lowest Common Denominator + sub-classes
- Requires a common/unifying model
  – Portable Document Format (PDF) for printed output
  – SCSI/SATA/SAS standard for disks, CDs, SSDs
- Usually involves a *federation framework*

Federation Frameworks

- A structure that allows many similar, but somewhat different, things to be treated uniformly
- By creating one interface that all must meet
- Then plugging in implementations for the particular things you have
- E.g., make all hard disk drives accept the same commands
  - Even though you have 5 different models installed

Are Federation Frameworks Too Limiting?

- Does the common model have to be the "lowest common denominator"?
- Not necessarily
  – The model can include "optional features",
    - Which (if present) are implemented in a standard way
    - But may not always be present (and can be tested for)
- Many devices will have features that cannot be exploited through the common model
  – There are arguments for and against the value of such features

Abstractions and Layering

- It's common to create increasingly complex services by layering abstractions
  - E.g., a generic file system layers on a particular file system, which layers on abstract disk, which layers on a real disk
- Layering allows good modularity
  - Easy to build multiple services on a lower layer
    - E.g., multiple file systems on one disk
  - Easy to use multiple underlying services to support a higher layer
  - E.g., file system can have either a single disk or a RAID below it

A Downside of Layering

- Layers typically add performance penalties
- Often expensive to go from one layer to the next
  - Since it frequently requires changing data structures or representations
  - At least involves extra instructions
- Another downside is that lower layer may limit what the upper layer can do
  - E.g., an abstract network link may hide causes of packet losses

Other OS Abstractions

- There are many other abstractions offered by the OS
- Often they provide different ways of achieving similar goals
  – Some higher level, some lower level
- The OS must do work to provide each abstraction
  – The higher level, the more work
- Programmers and users have to choose the right abstractions to work with

Conclusion

- Stable interfaces are critical to proper performance of an operating system
  - For program development (API)
  - For user experience (ABI)
- Abstractions make operating systems easier to use for both programmers and consumers
- The most important OS abstractions involve memory, interpreters, and communications