

Operating System Principles: Services, Resources, and Interfaces

CS 111

Operating Systems

Peter Reiher

Outline

- Operating systems services
- System service layers and mechanisms
- Service interfaces and standards
- Service and interface abstractions

Key OS Services

- Generally offered as abstractions
- Important basic categories:
 - CPU/Memory abstractions
 - Processes, threads, virtual machines
 - Virtual address spaces, shared segments
 - Persistent storage abstractions
 - Files and file systems
 - Other I/O abstractions
 - Virtual terminal sessions, windows
 - Sockets, pipes, VPNs, signals (as interrupts)

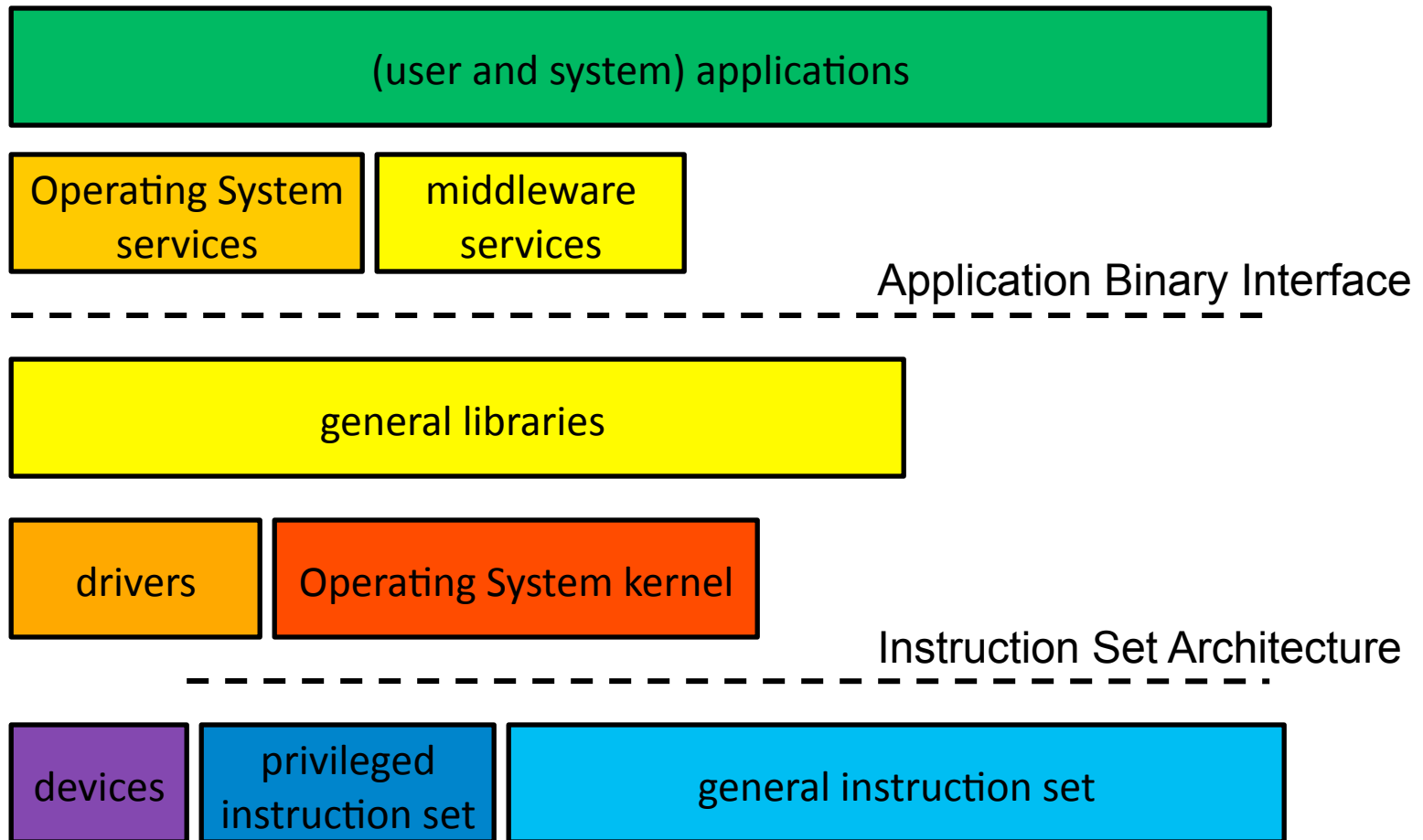
Services: Higher Level Abstractions

- Cooperating parallel processes
 - Locks, condition variables
 - Distributed transactions, leases
- Security
 - User authentication
 - Secure sessions, at-rest encryption
- User interface
 - GUI widgetry, desktop and window management
 - Multi-media

Services: Under the Covers

- Not directly visible to users
- Enclosure management
 - Hot-plug, power, fans, fault handling
- Software updates and configuration registry
- Dynamic resource allocation and scheduling
 - CPU, memory, bus resources, disk, network
- Networks, protocols and domain services
 - USB, BlueTooth
 - TCP/IP, DHCP, LDAP, SNMP
 - iSCSI, CIFS, NFS

Software Layering



How Can the OS Deliver These Services?

- Applications could just call subroutines
- Applications could make system calls
- Applications could send messages to software that performs the services
- At which layer does each of these options work?

Service Delivery via Subroutines

- Access services via direct subroutine calls
 - Push parameters, jump to subroutine, return values in registers on the stack
- Advantages
 - Extremely fast (nano-seconds)
 - DLLs enable run-time implementation binding
- Disadvantages
 - All services implemented in same address space
 - Limited ability to combine different languages
 - Can't usually use privileged instructions

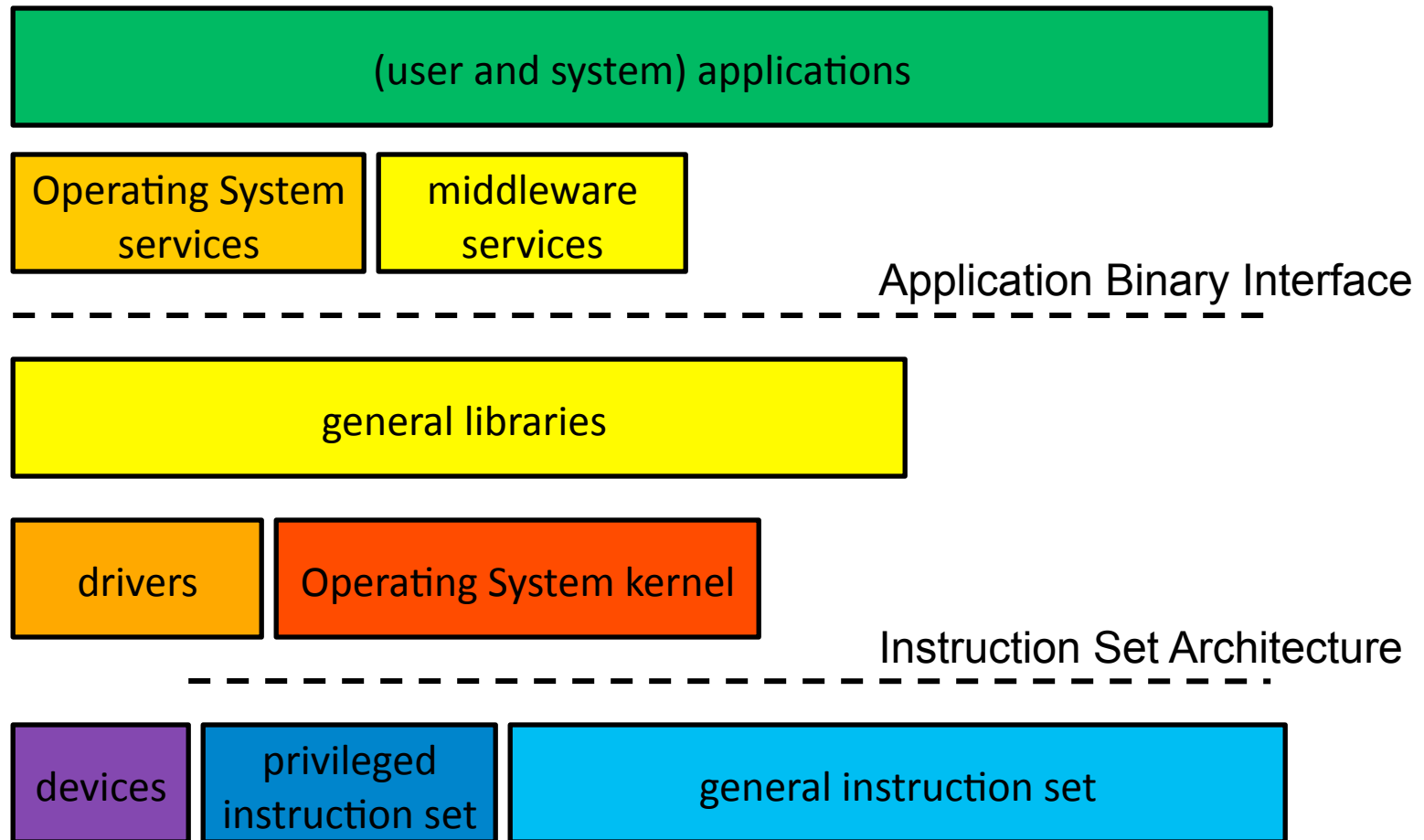
OS Layering

- Modern OSes offer services via layers of software and hardware
- High level abstract services offered at high software layers
- Lower level abstract services offered deeper in the OS
- Ultimately, everything mapped down to relatively simple hardware

Layers: Libraries

- Programmers need not write all code for programs
 - Standard utility functions can be found in libraries
- A library is a collection of object modules
 - A single file that contains many files (like a zip or jar)
 - These modules can be used directly, w/o recompilation
- Most systems come with many standard libraries
 - System services, encryption, statistics, etc.
 - Additional libraries may come with add-on products
- Programmers can build their own libraries
 - Functions commonly needed by parts of a product

The Library Layer



Characteristics of Libraries

- Many advantages
 - Reusable code makes programming easier
 - A single well written/maintained copy
 - Encapsulates complexity ... better building blocks
- Multiple bind-time options
 - Static ... include in load module at link time
 - Shared ... map into address space at exec time
 - Dynamic ... choose and load at run-time
- It is only code ... it has no special privileges

Shared Libraries

- Library modules are usually added to a program's load module
 - Each load module has its own copy of each library
 - This dramatically increases the size of each process
 - Program must be re-linked to incorporate new library
 - Existing load modules don't benefit from bug fixes
- Instead, make each library a sharable code segment
 - One in memory copy, shared by all processes
 - Keep the library separate from the load modules
 - Operating system loads library along with program

Advantages of Shared Libraries

- Reduced memory consumption
 - One copy can be shared by multiple processes/programs
- Faster program start-ups
 - If it's already in memory, it need not be loaded again
- Simplified updates
 - Library modules are not included in program load modules
 - Library can be updated (e.g., a new version with bug fixes)
 - Programs automatically get the newest version when they are restarted

Limitations of Shared Libraries

- Not all modules will work in a shared library
 - They cannot define/include global data storage
- They are read into program memory
 - Whether they are actually needed or not
- Called routines must be known at compile-time
 - Only the fetching of the code is delayed 'til run-time
 - Symbols known at compile time, bound at link time
- Dynamically Loadable Libraries are more general
 - They eliminate all of these limitations ... at a price

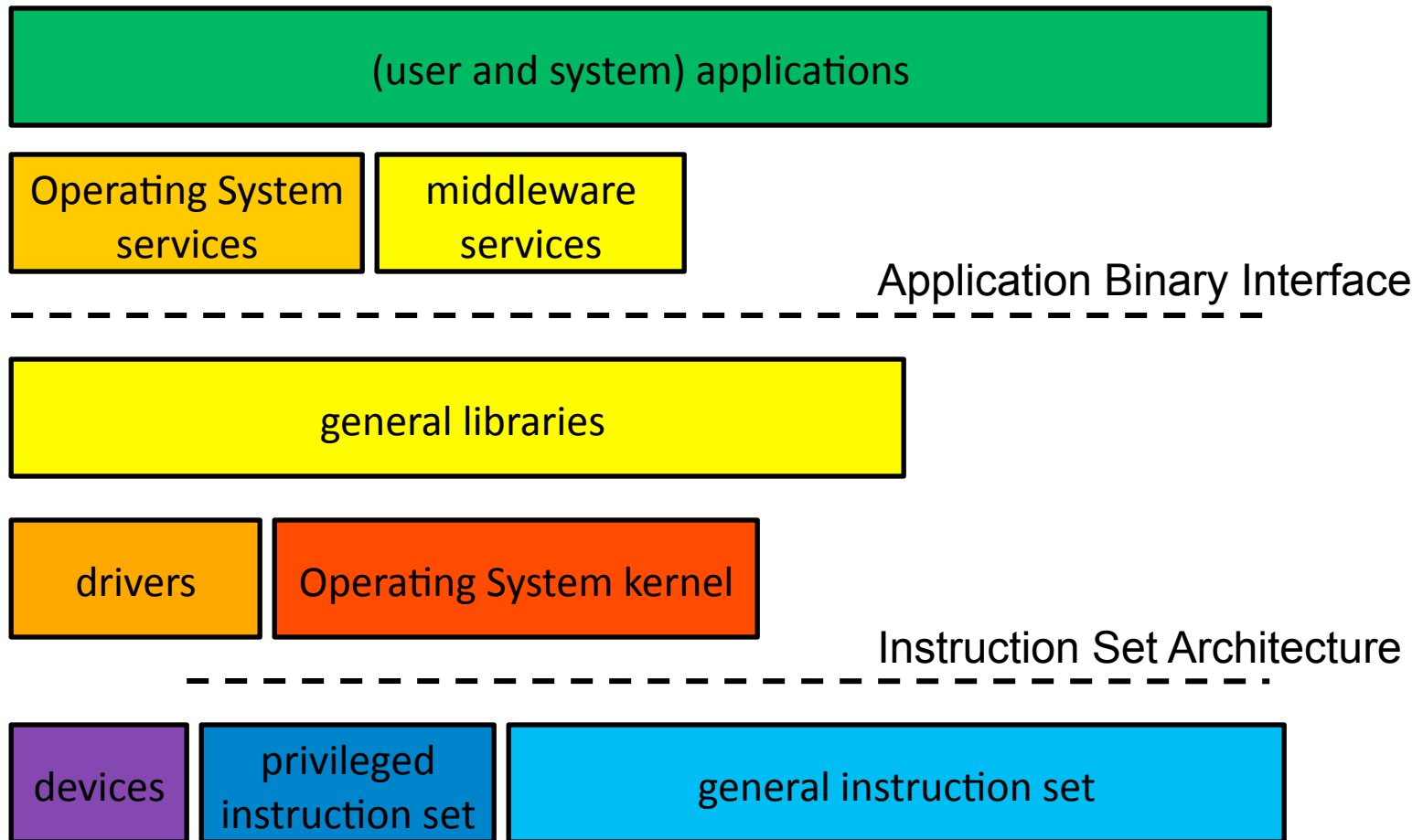
Service Delivery via System Calls

- Force an entry into the operating system
 - Parameters/returns similar to subroutine
 - Implementation is in shared/trusted kernel
- Advantages
 - Able to allocate/use new/privileged resources
 - Able to share/communicate with other processes
- Disadvantages
 - All implemented on the local node
 - 100x-1000x slower than subroutine calls

Layers: The Kernel

- Primarily functions that require privilege
 - Privileged instructions (e.g., interrupts, I/O)
 - Allocation of physical resources (e.g., memory)
 - Ensuring process privacy and containment
 - Ensuring the integrity of critical resources
- Some operations may be out-sourced
 - System daemons, server processes
- Some plug-ins may be less-trusted
 - Device drivers, file systems, network protocols

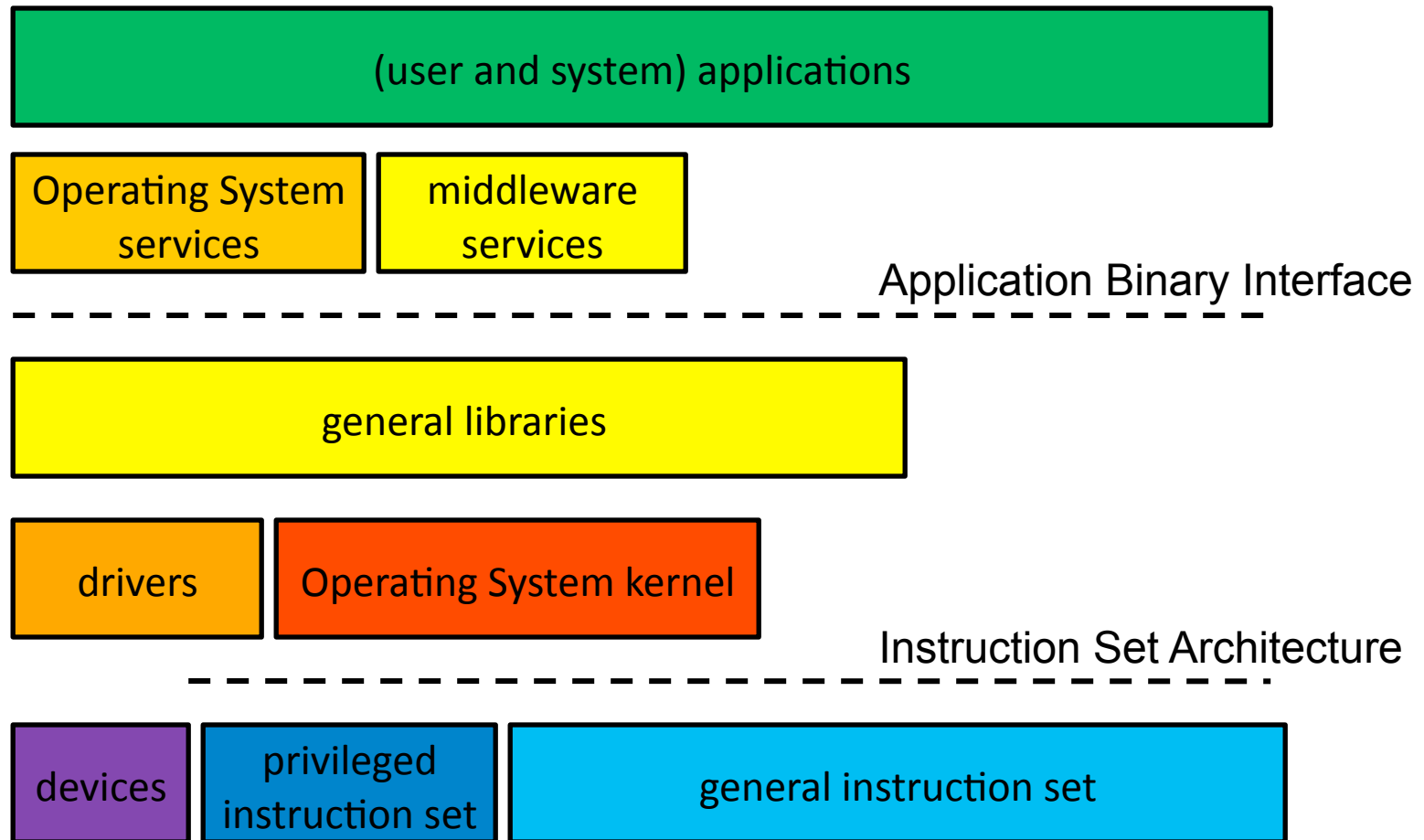
The Kernel Layer



Layers: System Services

- Not all trusted code must be in the kernel
 - It may not need to access kernel data structures
 - It may not need to execute privileged instructions
- Some are actually somewhat privileged processes
 - Login can create/set user credentials
 - Some can directly execute I/O operations
- Some are merely trusted
 - sendmail is trusted to properly label messages
 - NFS server is trusted to honor access control data

System Service Layer



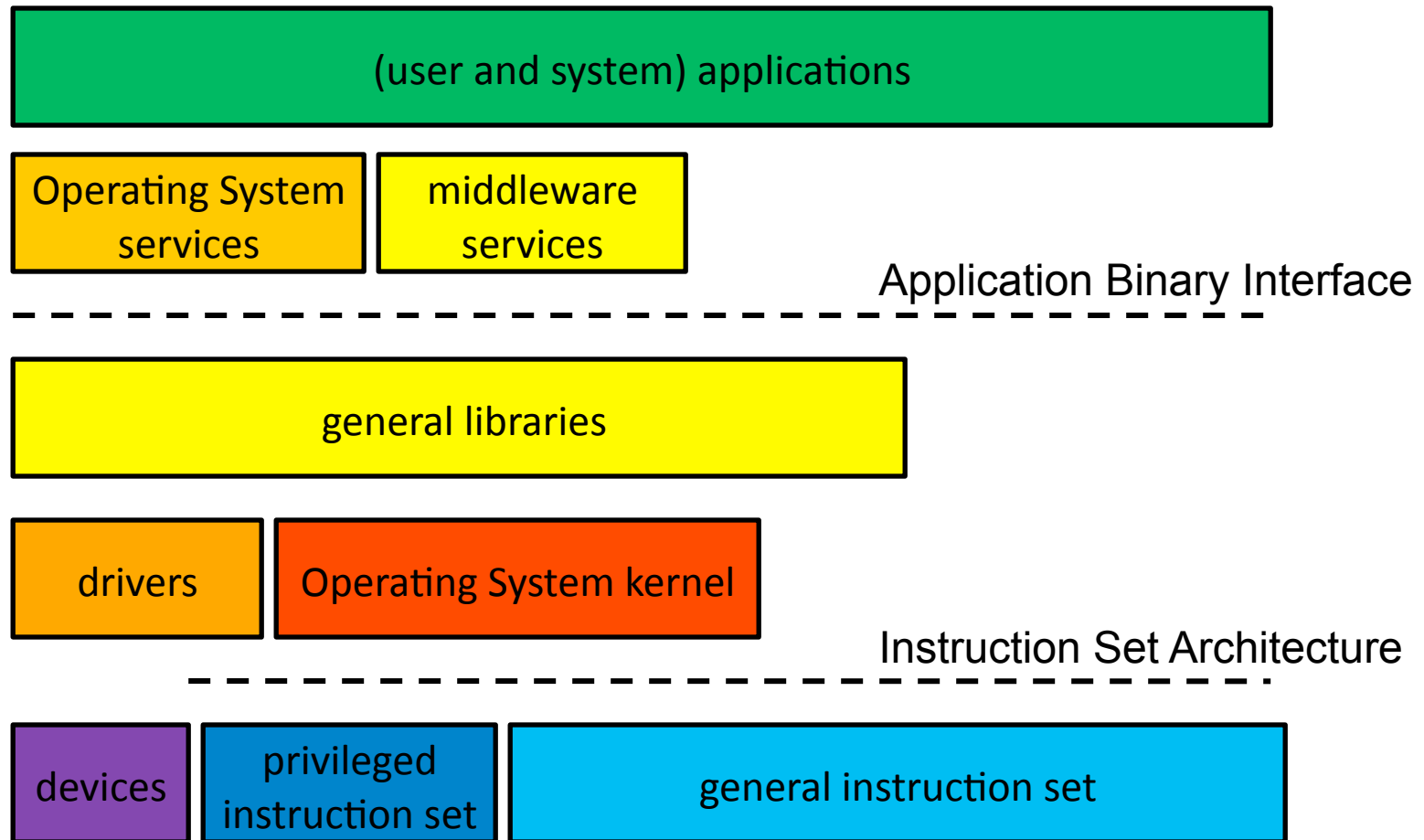
Service Delivery via Messages

- Exchange messages with a server (via syscalls)
 - Parameters in request, returns in response
- Advantages:
 - Server can be anywhere on earth
 - Service can be highly scalable and available
 - Service can be implemented in user-mode code
- Disadvantages:
 - 1,000x-100,000x slower than subroutine
 - Limited ability to operate on process resources

Layers: Middleware

- Software that is a key part of the application or service platform, but not part of the OS
 - Database, pub/sub messaging system
 - Apache, Nginx
 - Hadoop, Zookeeper, Beowulf, OpenStack
 - Cassandra, RAMCloud, Ceph, Gluster
- Kernel code is very expensive and dangerous
 - User-mode code is easier to build, test and debug
 - User-mode code is much more portable
 - User-mode code can crash and be restarted

The Middleware Layer



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

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

Why Does My OS Need to Support an ABI?

- Why not just support an API?
- Users would not like that much
- API-only compatibility requires them to obtain and compile their applications' sources
 - If it doesn't build, they have to debug it
- ABI compatibility allows merely loading and running the application (binary)
 - Of course, if it doesn't run, they're out of luck

User Mode Instruction Set vs. ABI

- Why distinguish the user mode instruction set from the Application Binary?
- The user mode instruction set is defined and implemented by hardware
 - It is thus ISA specific
- The Application Binary Interface is defined and implemented by software
 - It is thus OS specific
- Compilers generate code from the user-mode instruction set
- Code that exploits features in the Application Binary Interface is written by people (or higher level tools)

Other Important OS Interfaces

- Data formats and information encodings
 - Multi-media content (e.g. MP3, JPG)
 - Archival (e.g. tar, gzip)
 - File systems (e.g. DOS/FAT, ISO 9660)
- Protocols
 - Networking (e.g. ethernet, WLAN, TCP/IP)
 - Domain services (e.g. IMAP, LPD)
 - System management (e.g. DHCP, SNMP, LDAP)
 - Remote data access (e.g. FTP, HTTP, CIFS, S3)

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

Compatibility Taxonomy

- Upwards compatible (with ...)
 - New version still supports previous interfaces
- Backwards compatible (with ...)
 - Will correctly interact with old protocol versions
- Versioned interface, version negotiation
 - Parties negotiate a mutually acceptable version
- Compatibility layer
 - A cross-version translator
- Non-disruptive upgrade

Side Effects

- A *side effect* occurs when an action one object has non-obvious consequences
 - Perhaps even to other objects
 - Effects not specified by interfaces
- 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

Standards

- Different than interfaces
- Interfaces can differ from OS to OS
 - And machine to machine
- Standards are more global
- Either you follow a standard or you don't
 - If you do, others can work with you
 - If you don't, they can't
- Where did standards come from?

Standards in the Dark Ages (1965)

- No software industry as we now know it
- All the money was made on hardware
 - But hardware is useless without software
 - All software built by hardware suppliers
 - Platforms were distinguished by software
- Software portability was an anti-goal
 - Keep customers captive to your hardware
 - Portability means they could go elsewhere
- Standards were few and weak

The Software Reformation (1985)

- The advent of the "killer application"
 - Desk-top publishing, spreadsheets, ...
 - The rise of the Independent Software Vendor
- Fundamental changes to platform industry
 - The “applications, demand, volume” cycle
 - Application capture became strategic
- Applications portability became strategic
 - Standards are the key to portability
 - Standards compliance became strategic

The Role of Standards Today

- There are many software standards
 - Subroutines, protocols and data formats, ...
 - Both portability and interoperability
 - Some are general (e.g. POSIX 1003, TCP/IP)
 - Some are very domain specific (e.g. MPEG2)
- Key standards are widely required
 - Non-compliance reduces application capture
 - Non-compliance raises price to customers
- Bottom line: if you don't meet the standard, your system isn't used

Where Do Standards Stop?

- Why not just one browser for everyone?
- And just one image format?
- And just one email program?
- Those could be standards themselves
- Why not?
- Why not just bundle everything into the OS?



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

Abstractions: An Object-Oriented View

- My execution platform implements objects
 - They may be bytes, longs, and strings
 - They may be processes, files, and sessions
- An *object* is defined by
 - Its properties, methods, and their semantics
- What makes a particular set of objects good?
 - They are powerful enough to do what I need
 - They don't force me to do a lot of extra work
 - They are simple enough for me to understand

Simplifying Abstractions

- Hardware is fast, but complex and limited
 - Using it correctly is extremely complex
 - It may not support the desired functionality
 - It is not a solution, but merely a building block
- Encapsulate implementation details
 - Error handling, performance optimization
 - Eliminate behavior that is irrelevant to the user
- 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

The Basic Memory Operations

- Regardless of level or type, memory abstractions support a couple of operations
 - WRITE(name, value)
 - Put a value into a memory location specified by name
 - value <- READ(name)
 - Get a value out of a memory location specified by name
- Seems pretty simple
- But going from a nice abstraction to a physical implementation can be complex

Some Complicating Factors

- Persistent vs. transient memory
- Size of 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 without 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*
- If there are several reads/writes to the file, we expect each to occur in some order
 - With respect to the others

What Is Implementing the File?

- Commonly a hard disk drive
- Disk drives have peculiar characteristics
 - Long, and worse, variable access latencies
 - Accesses performed in chunks of fixed size
 - Atomicity only for accesses of that size
 - Highly variable performance depending on exactly what gets put where
 - Unpleasant failure modes
- So the operating system needs to smooth out these oddities

What Does That Lead To?

- Great effort by file system component of OS to put things in the right place on a disk
- Reordering of disk operations to improve performance
 - Which complicates providing atomicity
- Optimizations based on caching and read-ahead
 - Which complicates maintaining consistency
- Sophisticated organizations to handle failures

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 almost always are multiple processes
- The OS has a certain amount of physical memory
 - To hold the environment information
- There is usually only one set of registers
- The process doesn't have exclusive access to the CPU
 - Due to 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

Basic Communication Link Operations

- **SEND(link_name, outgoing_message_buffer)**
 - Send some information contained in the buffer on the named link
- **RECEIVE(link_name, incoming_message_buffer)**
 - Read some information off the named link and put it into the buffer
- Like **WRITE** and **READ**, in some respects

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

An Example Communications Link

- A Unix-style socket
- SEND interface:
 - `send(int sockfd, const void *buf, size_t len, int flags)`
 - The `sockfd` is the link name
 - The `buf` is the outgoing message buffer
- RECEIVE interface:
 - `recv(int sockfd, void *buf, size_t len, int flags)`
 - Same parameters as for `send`

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 to perform the transfer
 - Either copy the 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 Are the Implications?

- Greater uncertainty about the outcome of an operation
 - Things fail for reasons our OS can't see or learn
 - Even on local machine, since OS doesn't control most of receiver's behavior
- Greater asynchrony
 - Even on a single machine
- Higher possibilities for security problems
 - Particularly when receiver is remote
 - You've heard that a lot

What Do We Do About Those Issues?

- OS must be prepared for likely failures
- And high degrees of asynchrony
 - Bad idea to block entire system while waiting for message delivery
- OS shouldn't have complete trust in what comes in from the network
 - But often the OS is in no position to determine its trustworthiness

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* 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 file system layers on top of an abstract disk, which layers on top of 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 disk prevents disk operation reorderings to maximize performance

Layer Bypassing

- Often necessary to allow a high layer to access much lower layers
 - Not going through one or more intermediaries
- Most commonly for performance reasons
- If the higher layer plans to use the very low level layer's services,
 - Why pay the cost of the intermediate layer?
- Layer bypassing has its downsides, too
 - Intermediate layer can't help or understand

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