Operating System Abstractions and Services
CS 111
Summer 2025
Operating System Principles

Peter Reiher

Outline

- What is an OS abstraction?
- What are important OS abstraction types?
- What is an OS service?
- How can we offer OS services?

The OS and Abstraction

- One major function of an OS is to offer abstract versions of resources
  – As opposed to actual physical resources
- Essentially, the OS implements the abstract resources using the physical resources
  – E.g., processes (an abstraction) are implemented using the CPU and RAM (physical resources)
  – And files (an abstraction) are implemented using flash drives (a physical resource)

Why Abstract Resources?

- The abstractions are typically simpler and better suited for programmers and users
  - Easier to use than the original resources
    - E.g., don't need to worry about keeping track of disk interrupts
  - Compartmentalize/encapsulate complexity
    - E.g., need not be concerned about what other executing code is doing and how to stay out of its way
  - Eliminate behavior that is irrelevant to user
    - E.g., hide the slow erase cycle of flash memory
  - Create more convenient behavior
    - E.g., make it look like you have the network interface entirely for your own use

Generalizing Abstractions

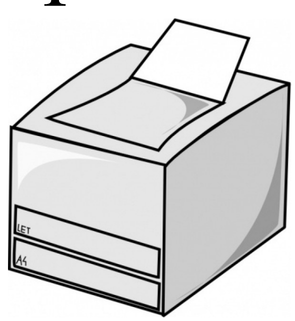
- Lots of variations in machines' HW and SW
- Make many different types appear the same
  - So applications can deal with single common class
- Usually involves a common unifying model
  - E.g., portable document format (pdf) for printers
  - Or SCSI standard for disks, CDs and tapes
- For example:
  - Printer drivers make different printers look the same
  - Browser plug-ins to handle multi-media data

Common Types of OS Resources

- Serially reusable resources
- Partitionable resources
- Sharable resources

Serially Reusable Resources

- Used by multiple clients, but only one at a time
  - Time multiplexing
- Require access control to ensure exclusive use
- Require graceful transitions from one user to the next
- Examples:

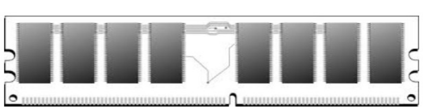


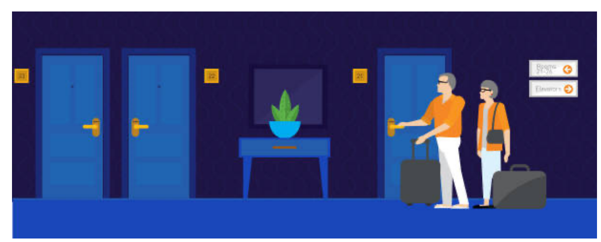
What Is A Graceful Transition?

- A switch that totally hides the fact that the resource used to belong to someone else
  - Don’t allow the second user to access the resource until the first user is finished with it
    - No incomplete operations that finish after the transition
  - Ensure that each subsequent user finds the resource in “like new” condition
    - No traces of data or state left over from the first user

Partitionable Resources

- Divided into disjoint pieces for multiple clients
  – Spatial multiplexing
- Needs access control to ensure:
  – Containment: *you cannot access resources outside of your partition*
  – Privacy: *nobody else can access resources in your partition*
- Examples:





Do We Still Need Graceful Transitions?

- Yes
- Most partitionable resources aren’t permanently allocated
  - The piece of RAM you’re using now will belong to another process later
- As long as it’s “yours,” no transition required
- But sooner or later it’s likely to become someone else’s

Shareable Resources

- Usable by multiple concurrent clients
  - Clients don’t “wait” for access to resource
  - Clients don’t “own” a particular subset of the resource
- May involve (effectively) limitless resources
  - Air in a room, shared by occupants
  - Copy of the operating system, shared by processes

Do We Still Need Graceful Transitions?

- Typically not
- The shareable resource usually doesn't change state
- Or isn't "reused"
- We never have to clean up what doesn't get dirty
  - Like an execute-only copy of the OS
- Shareable resources are great!
  - When you can have them . . .

Tip: Design your system to maximize sharable resources.

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?

- On a given machine, the OS doesn't have abstract devices with arbitrary properties
- It has particular physical devices specific to <u>that</u> machine
  - With unchangeable, often inconvenient, properties
  - Different devices on other machines
- The core OS abstraction problem:
  - Creating the abstract device with the desirable properties from physical devices 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

OS Services

- The operating system offers important services to other programs
- 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)

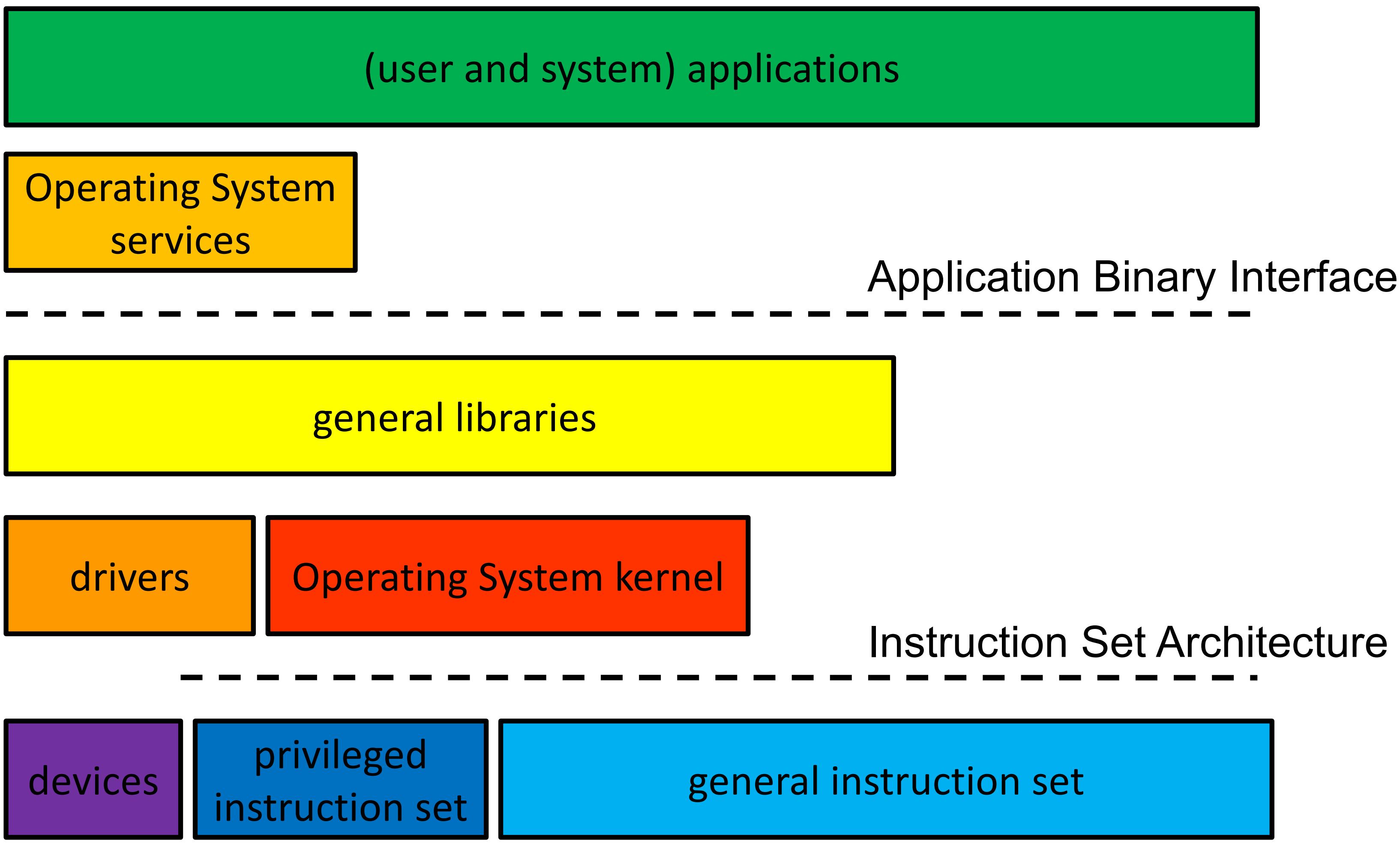
How Can the OS Deliver These Services?

- Several possible ways
  - Applications could just call subroutines
  - Applications could make system calls
- Each option works at a different *layer* of the stack of software

OS Layering

- Modern OSes offer services via layers of software and hardware
- High level abstract services offered at high software layers
  - Today, often “OS on another OS”
- Lower level abstract services offered deeper in the OS
- But ultimately, everything is mapped down to relatively simple hardware

Software Layering



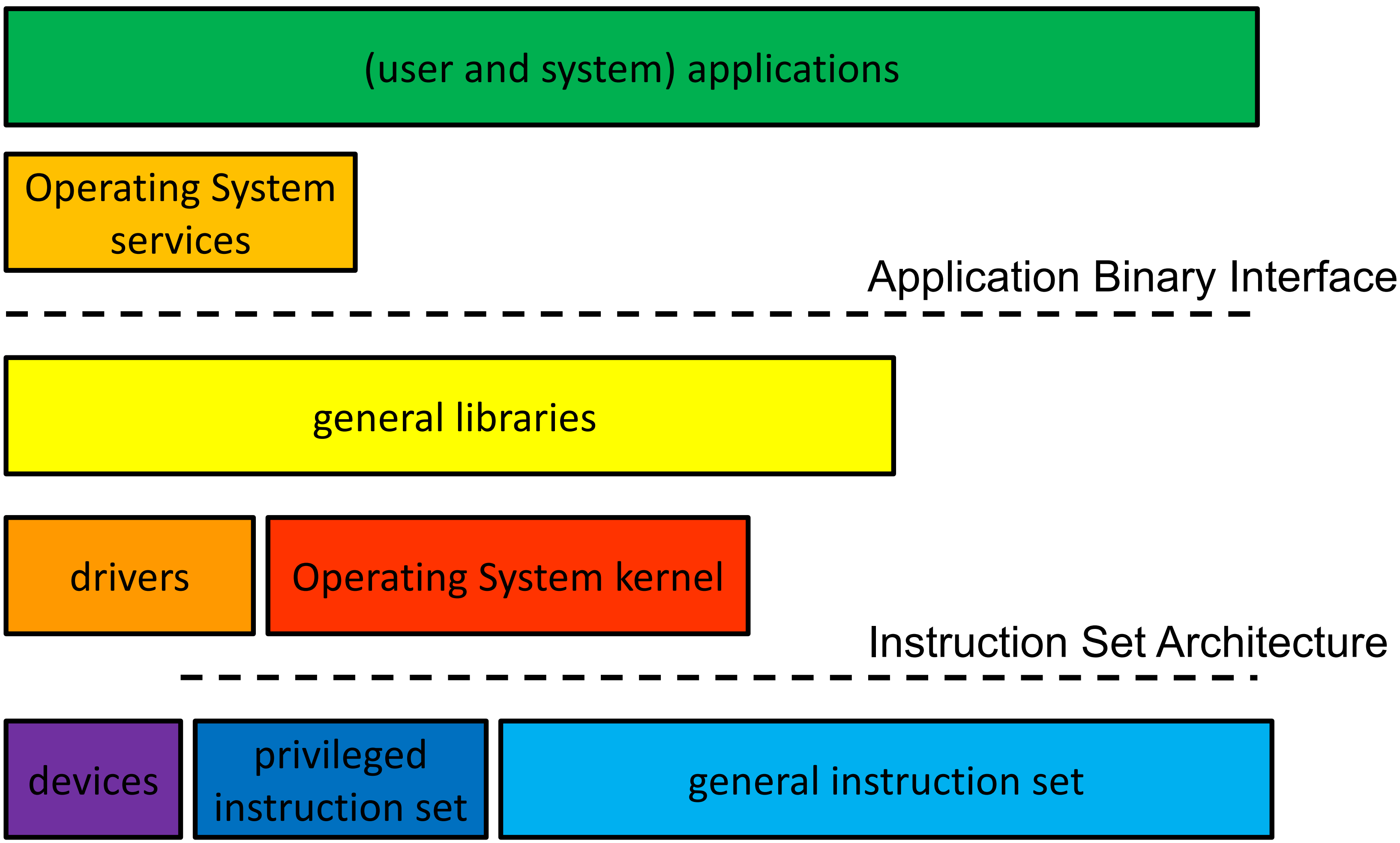
Service Delivery via Subroutines

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

Service Delivery via Libraries

- One subroutine service delivery approach
- 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

Sharing Libraries

- *Static library* modules are 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

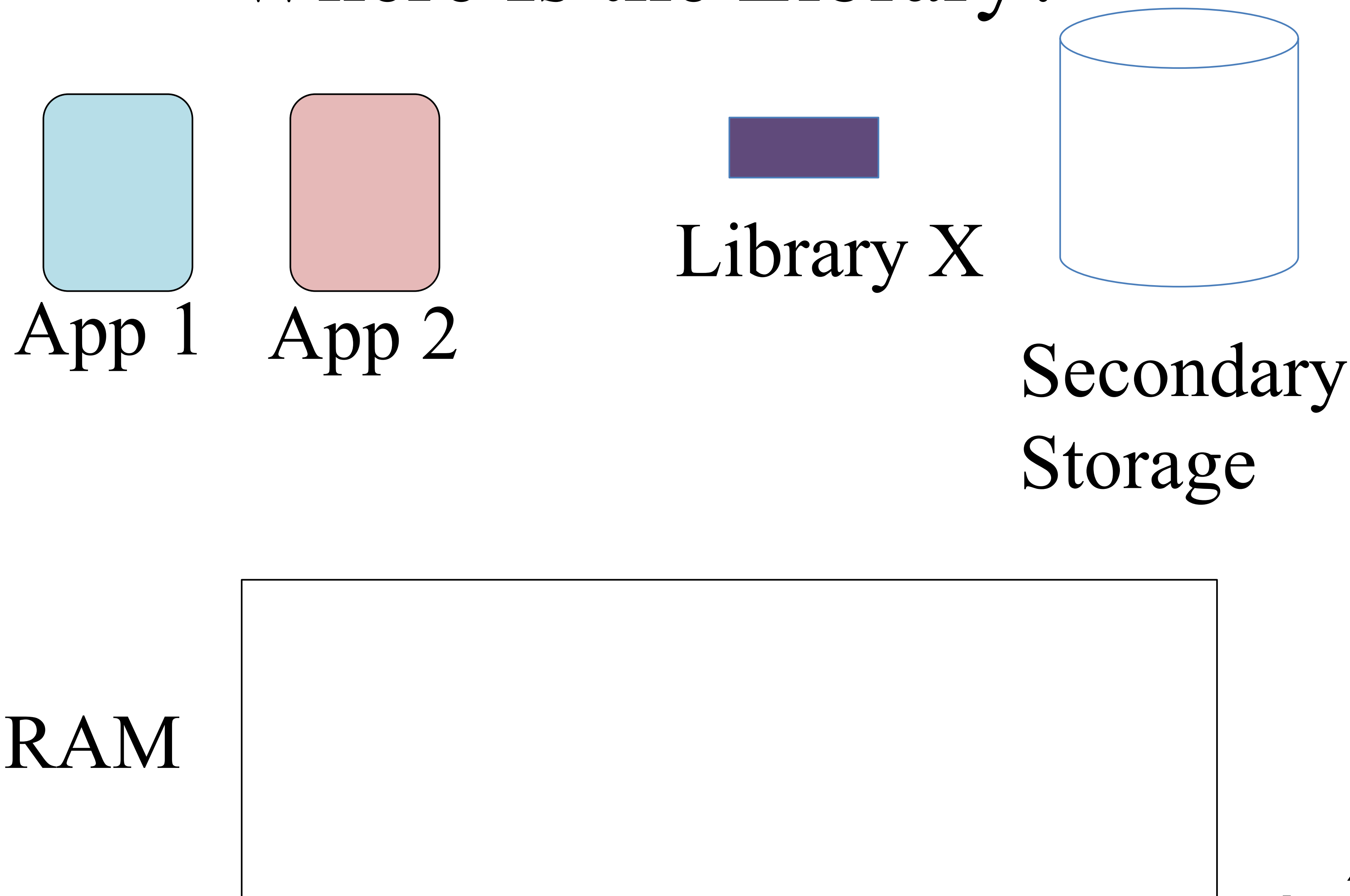
Advantages of Shared Libraries

- 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 easily (e.g., a new version with bug fixes)
  - Programs automatically get the newest version when they are restarted

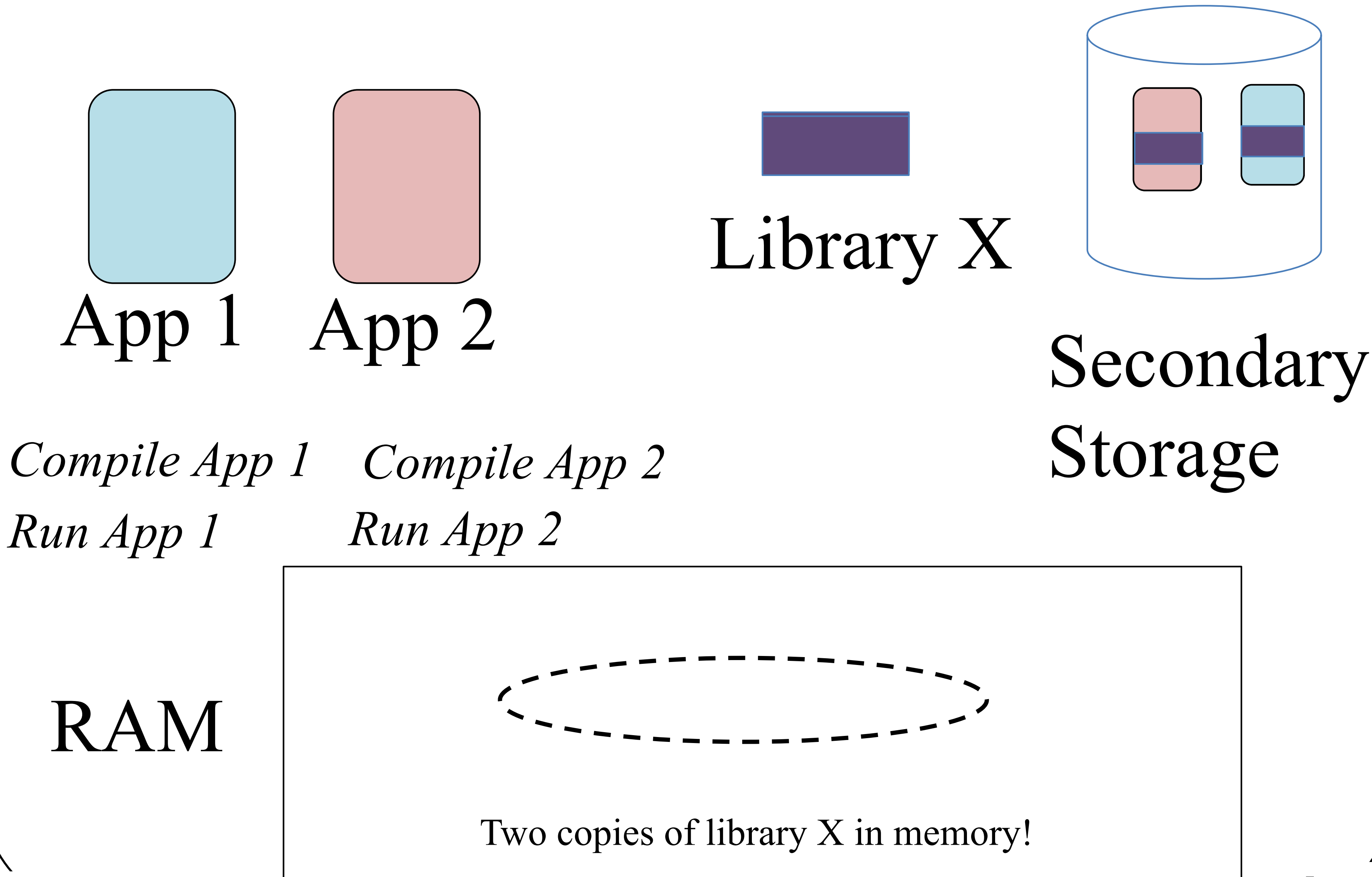
Limitations of Shared Libraries

- Not all modules will work in a shared library
  - They cannot define/include global data storage
- They are added 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

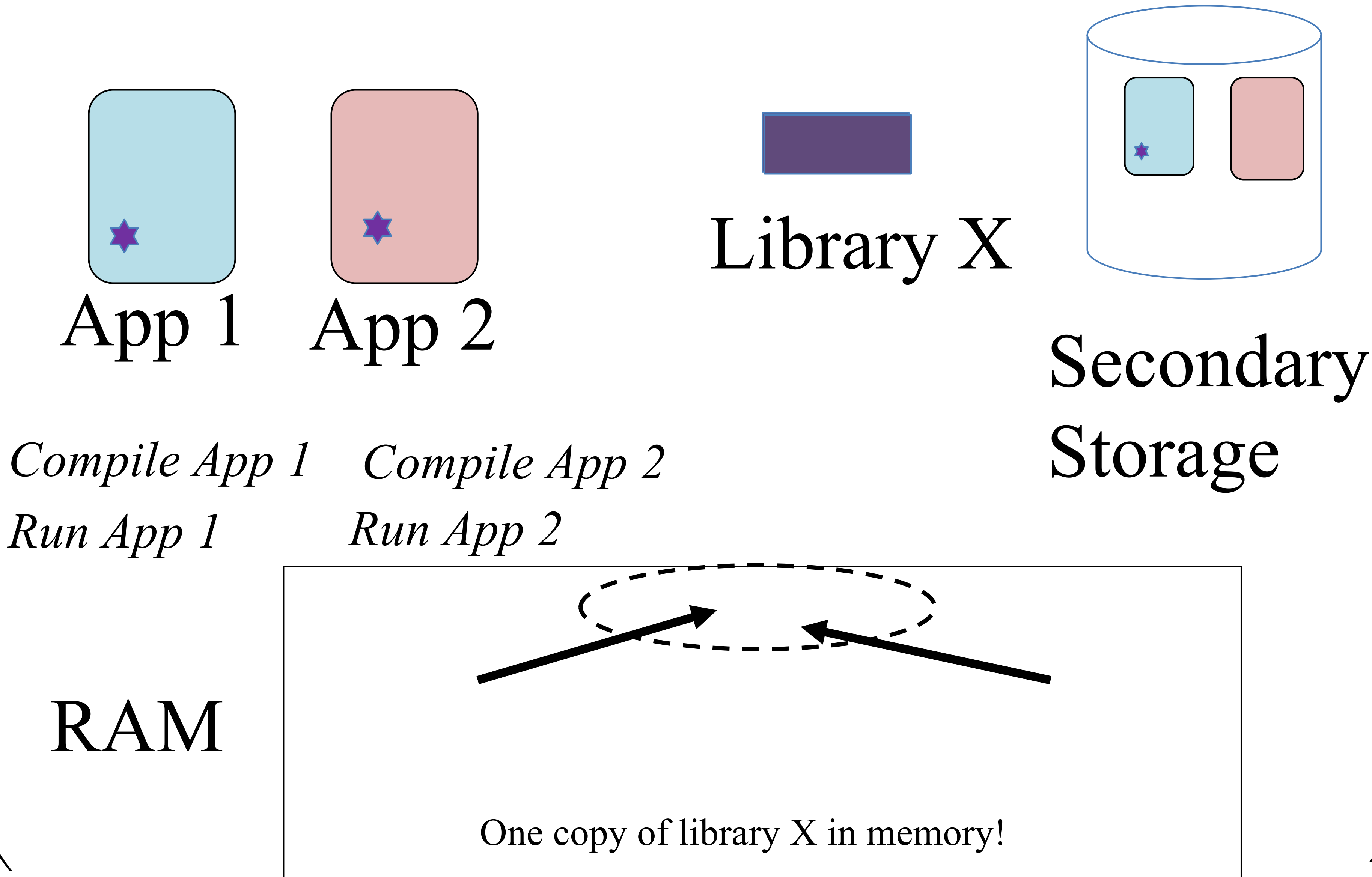
Where Is the Library?



Static Libraries



Shared Libraries



Dynamic Libraries

App 1 App 2

Library X

Secondary Storage

Compile App 1 *Compile App 2*

Run App 1 *App 1 uses the dynamic library*

RAM

Load only the dynamic libraries that are used

At the moment when they are requested

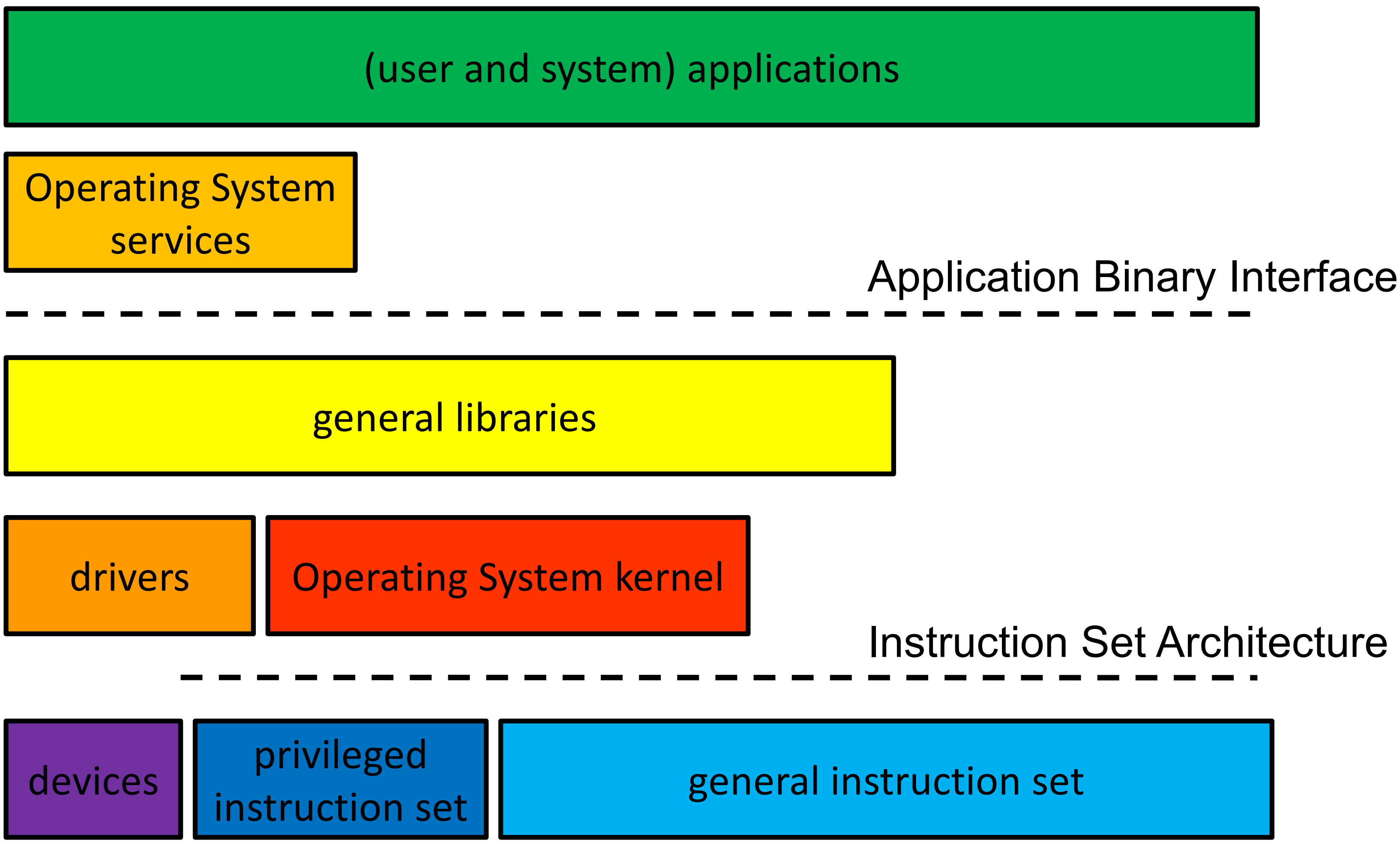
Service Delivery via System Calls

- 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
  - 100x-1000x slower than subroutine calls

Providing Services via 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

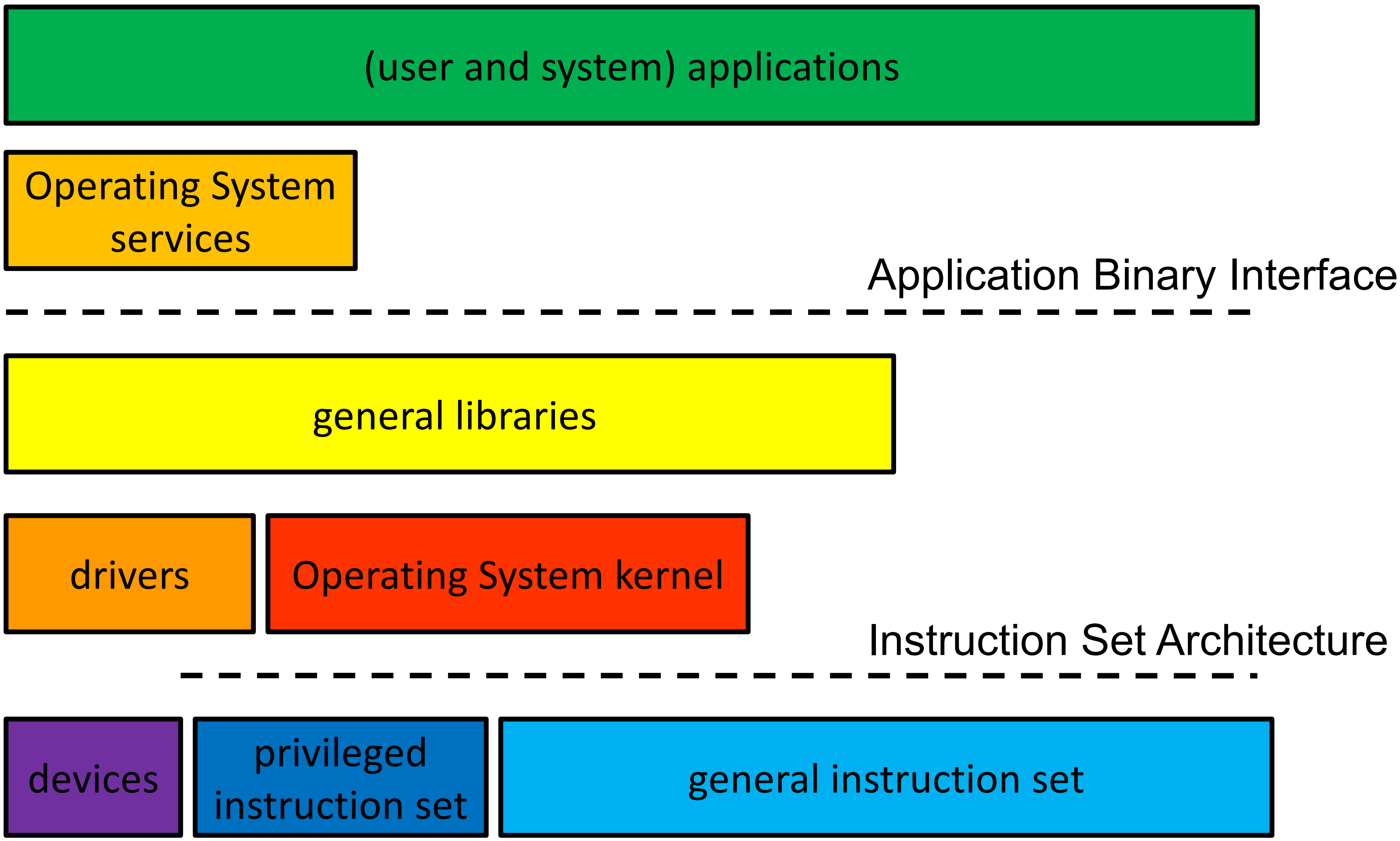
The Kernel Layer



System Services Outside the Kernel

- 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



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
- How can users and programs access those services?
- *Interfaces* are required between the OS and other programs to offer 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 <u>write</u> 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 to help users

ABIs help you install binaries on your OS

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

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!

Conclusion

- Operating systems provide services via abstractions
- Operating systems offer services at several layers in the software stack
- OS services are accessed via well-defined and stable interfaces
  - The API for program development
  - The ABI for easy distribution of software