Operating System Services
CS 111
Winter 2023
Operating System Principles
Peter Reiher

Outline

- Operating systems and abstractions
- Trends in operating systems
- Operating system 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

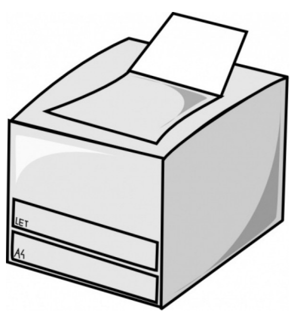
- Lots of variations in machines' HW and SW
- Make many different types appear the same
  - So applications can deal with single common class
- Usually involves a common unifying model
  - E.g., portable document format (pdf) for printers
  - Or SCSI standard for disks, CDs and tapes
- For example:
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

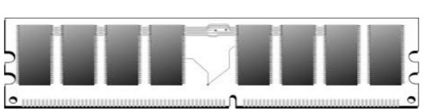


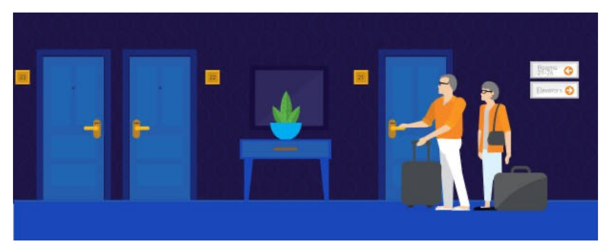
What Is A Graceful Transition?

- 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.

General OS Trends

- They have grown larger and more sophisticated
- Their role has fundamentally changed
  - From shepherding the use of the hardware
  - To shielding the applications from the hardware
  - To providing powerful application computing platform
  - To becoming a sophisticated "traffic cop"
- They still sit between applications and hardware
- Best understood through services they provide
  - Capabilities they add
  - Applications they enable
  - Problems they eliminate

Why?

- Ultimately because it's what users want
- The OS must provide core services to applications
- Applications have become more complex
  - More complex internal behavior
  - More complex interfaces
  - More interactions with other software
- The OS needs to help with all that complexity

OS Convergence

What about Chrome OS?

- There are a handful of widely used OSes

Windows 1985 | Mac OS 1984 | Linux 1991

And a few special purpose ones (e.g., real time and embedded system OSes) FreeBSD QNX TinyOS

- OSes in the same family are used for vastly different purposes
  - Challenging for the OS designer

It's Linux-based.

- Most OSes are based on pretty old models

Why Have OSes Converged?

- They're expensive to build and maintain
  - So it's a hard business to get into and stay in
- They only succeed if users choose them over other OS options
  - Which can't happen unless you support all the apps the users want
  - Which requires other parties to do a lot of work
- You need to have some clear advantage over present acceptable alternatives

Where Are The Popular OSes Used?

- Windows
  - The most popular choice for personal computers
  - Laptops, desktops, etc.
  - Some use in servers and small devices
- MacOS
  - Exclusively in Apple products
  - But in <u>all</u> Apple products (Macbooks, iPhones, Apple Watches, etc.)
- Linux
  - The choice in industrial servers (e.g., cloud computing)
  - And the choice of CS nerds and embedded systems

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)

Services: Higher Level Abstractions

- Cooperating parallel processes
  - Locks, condition variables
  - Distributed transactions, leases
- Security
  - User authentication
  - Secure sessions, at-rest encryption
- User interface
  - GUI widgets, 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

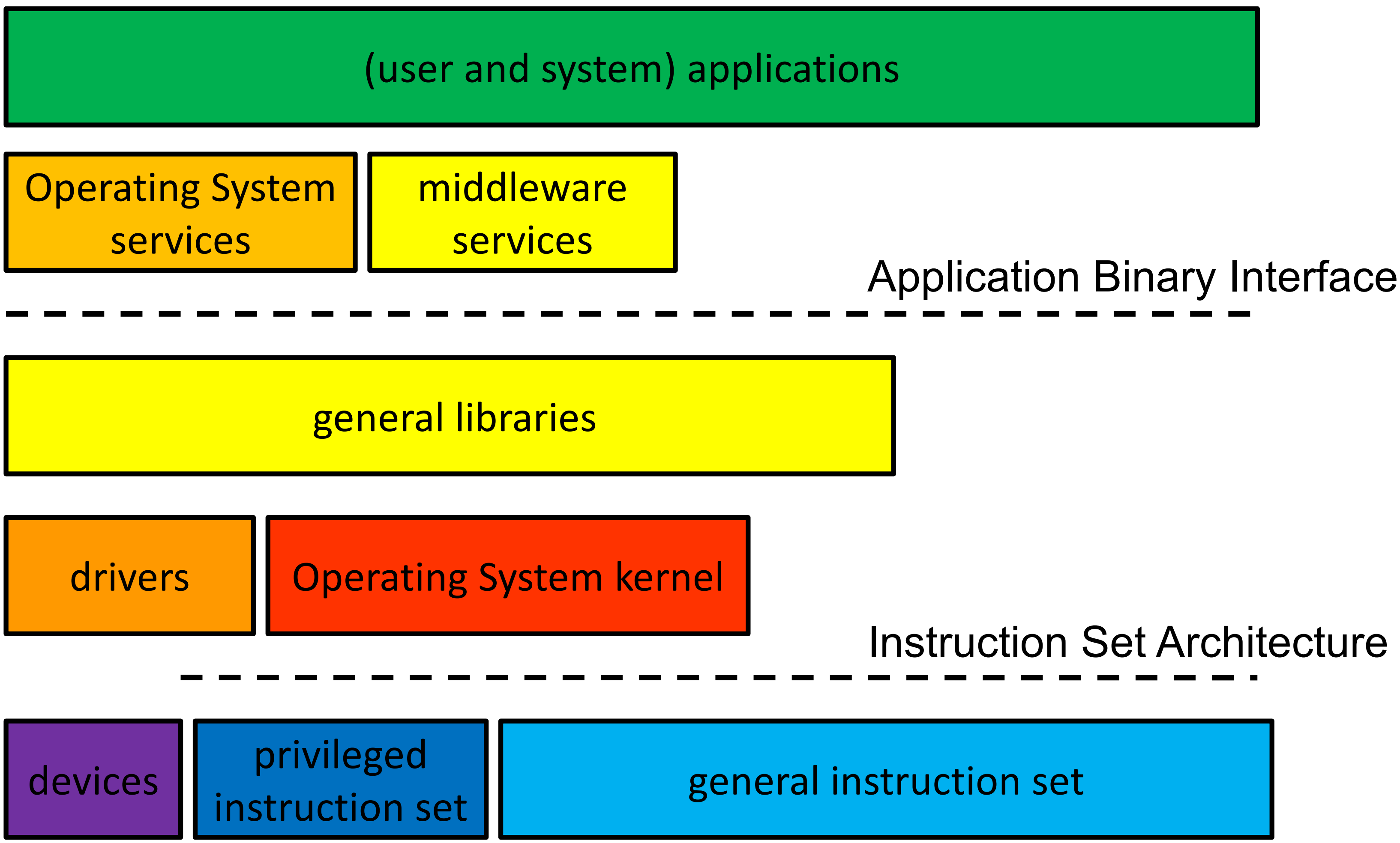
How Can the OS Deliver These Services?

- Several possible ways
  - Applications could just call subroutines
  - Applications could make system calls
  - Applications could send messages to software that performs the services
- 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
- Lower level abstract services offered deeper in the OS
- Ultimately, everything mapped down to relatively simple hardware

Software Layering



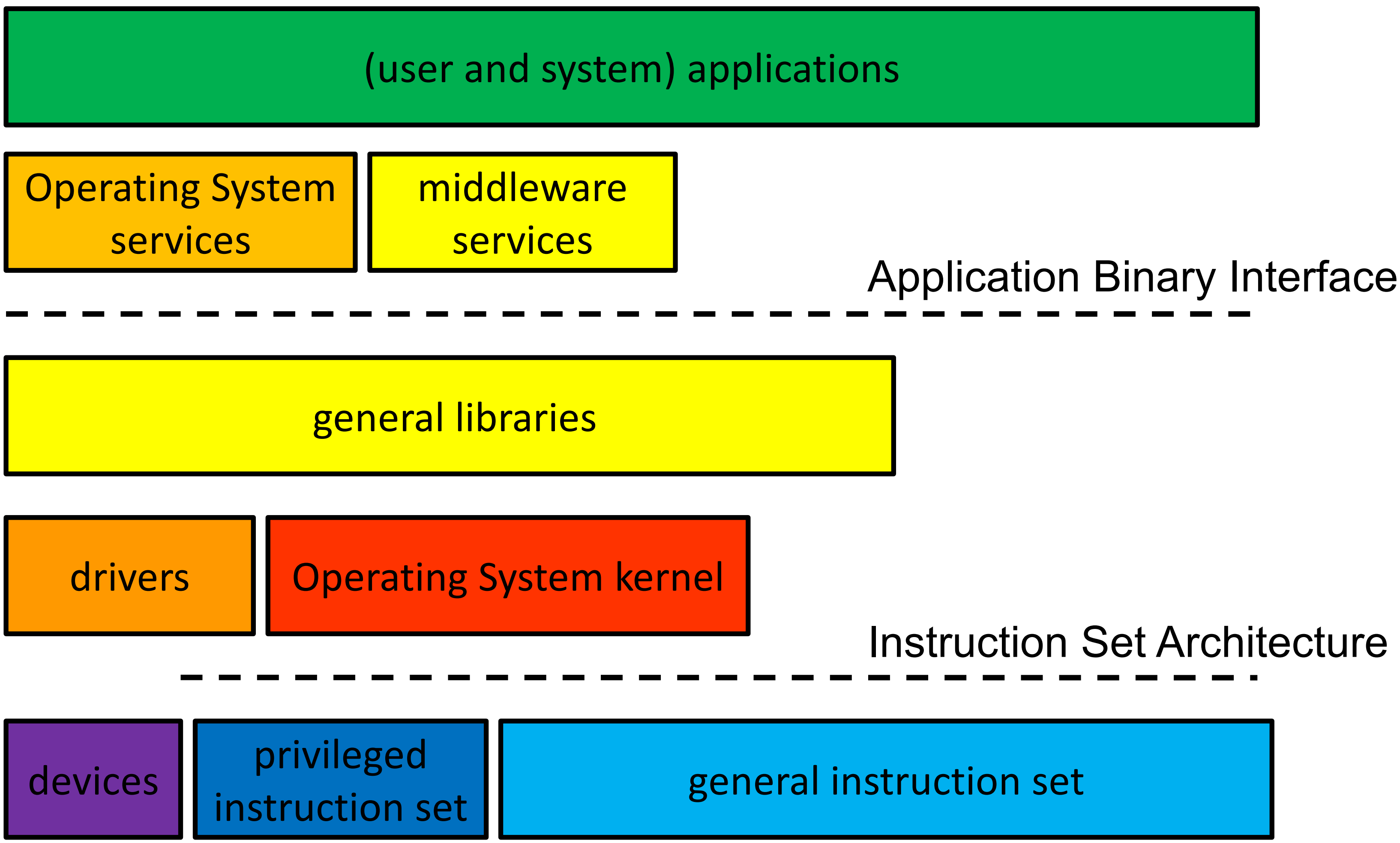
Service Delivery via Subroutines

- 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

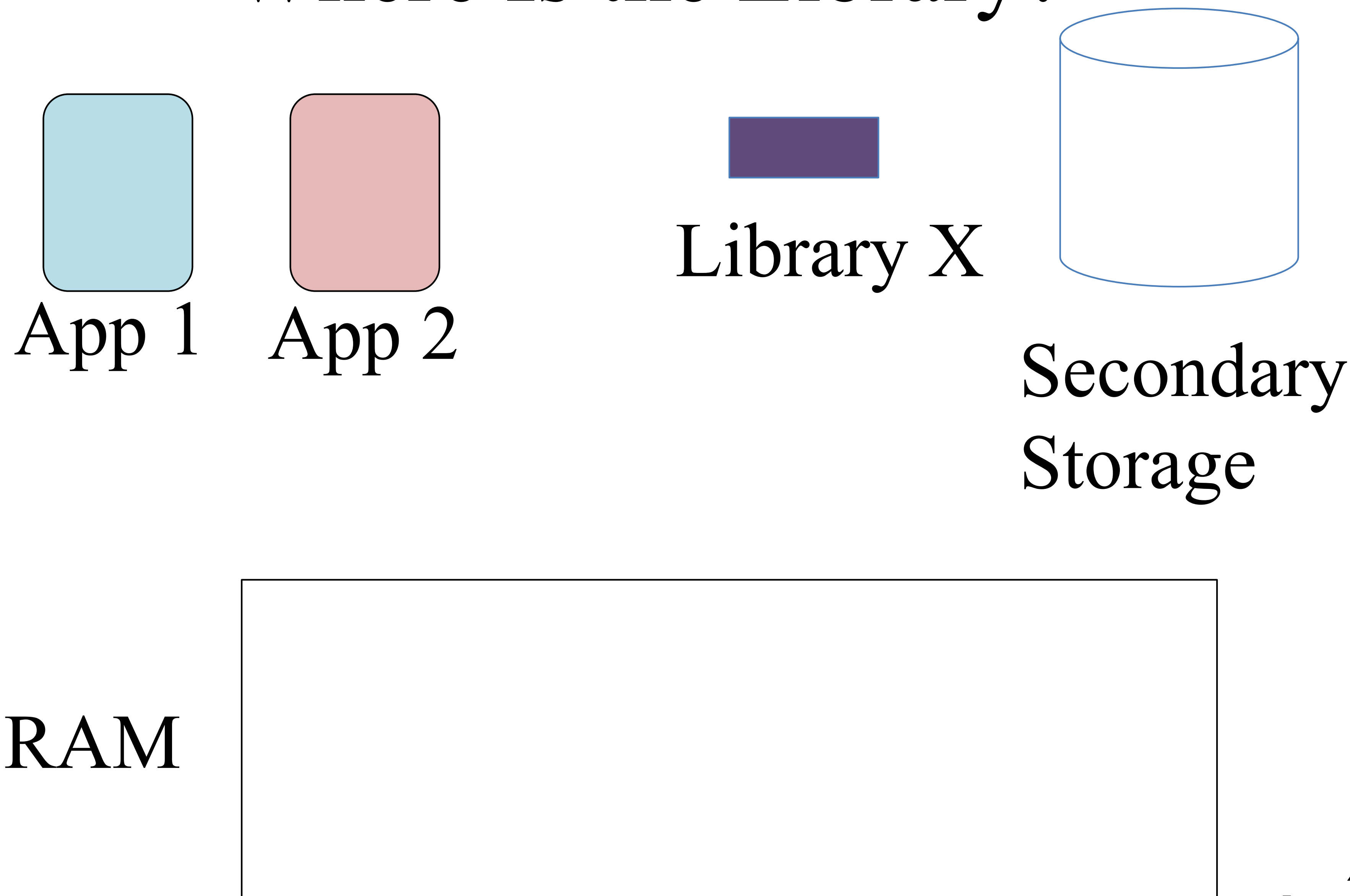
Advantages of Shared Libraries

- 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

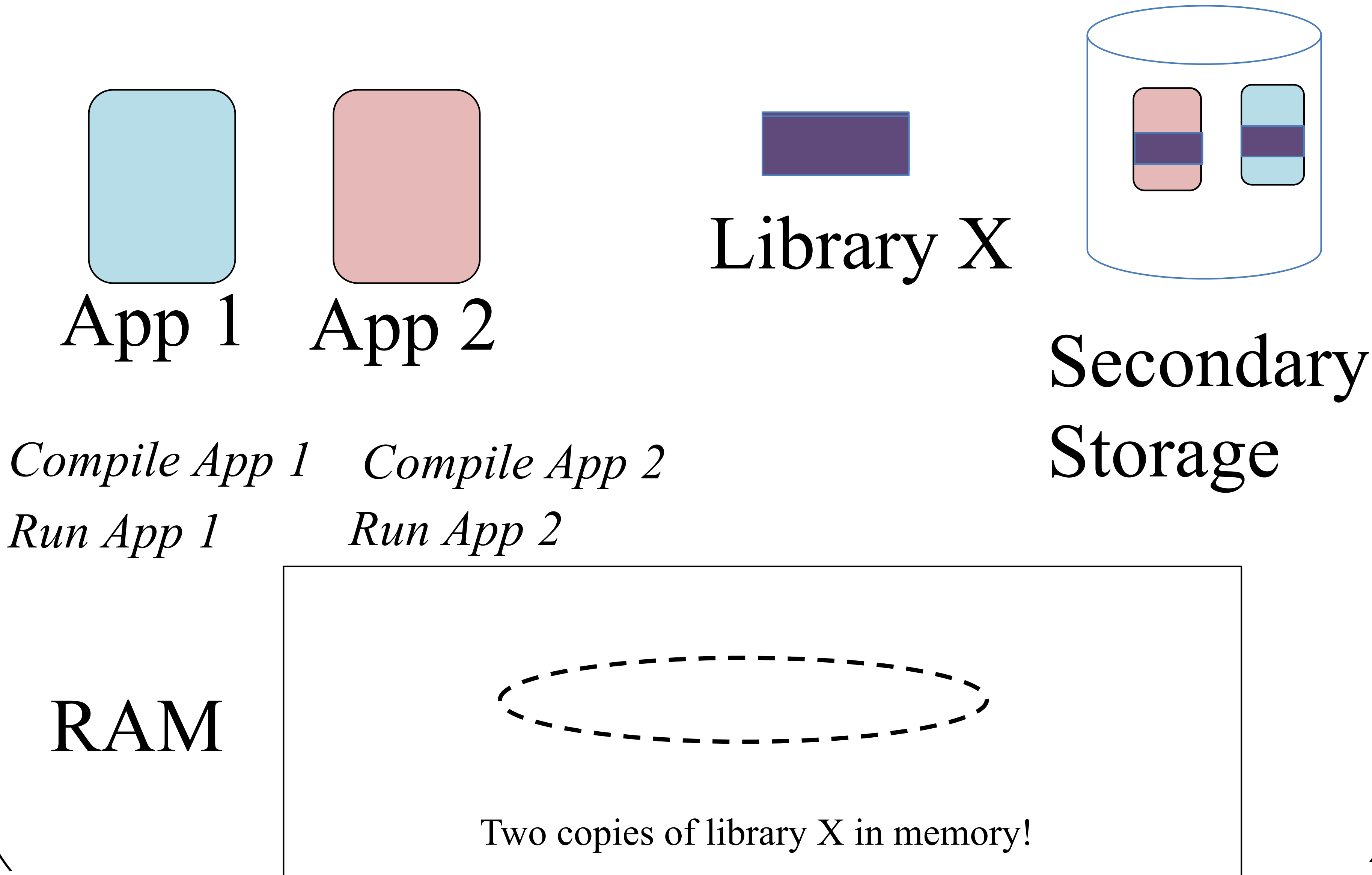
Limitations of Shared Libraries

- 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

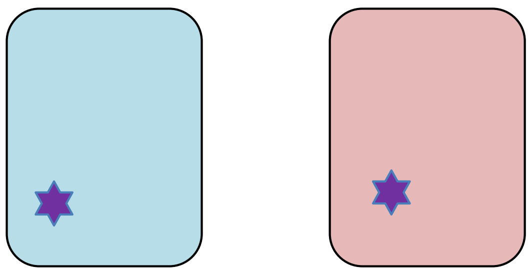
Where Is the Library?



Static Libraries

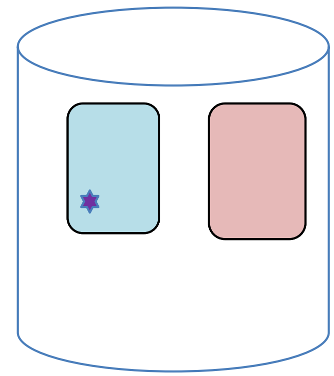


Shared Libraries



App 1 App 2

Library X

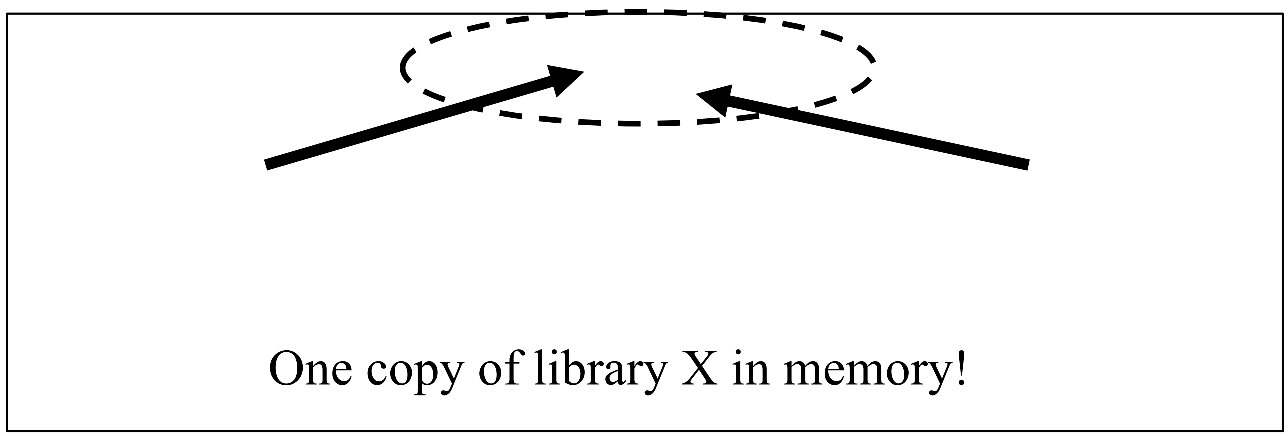


Secondary Storage

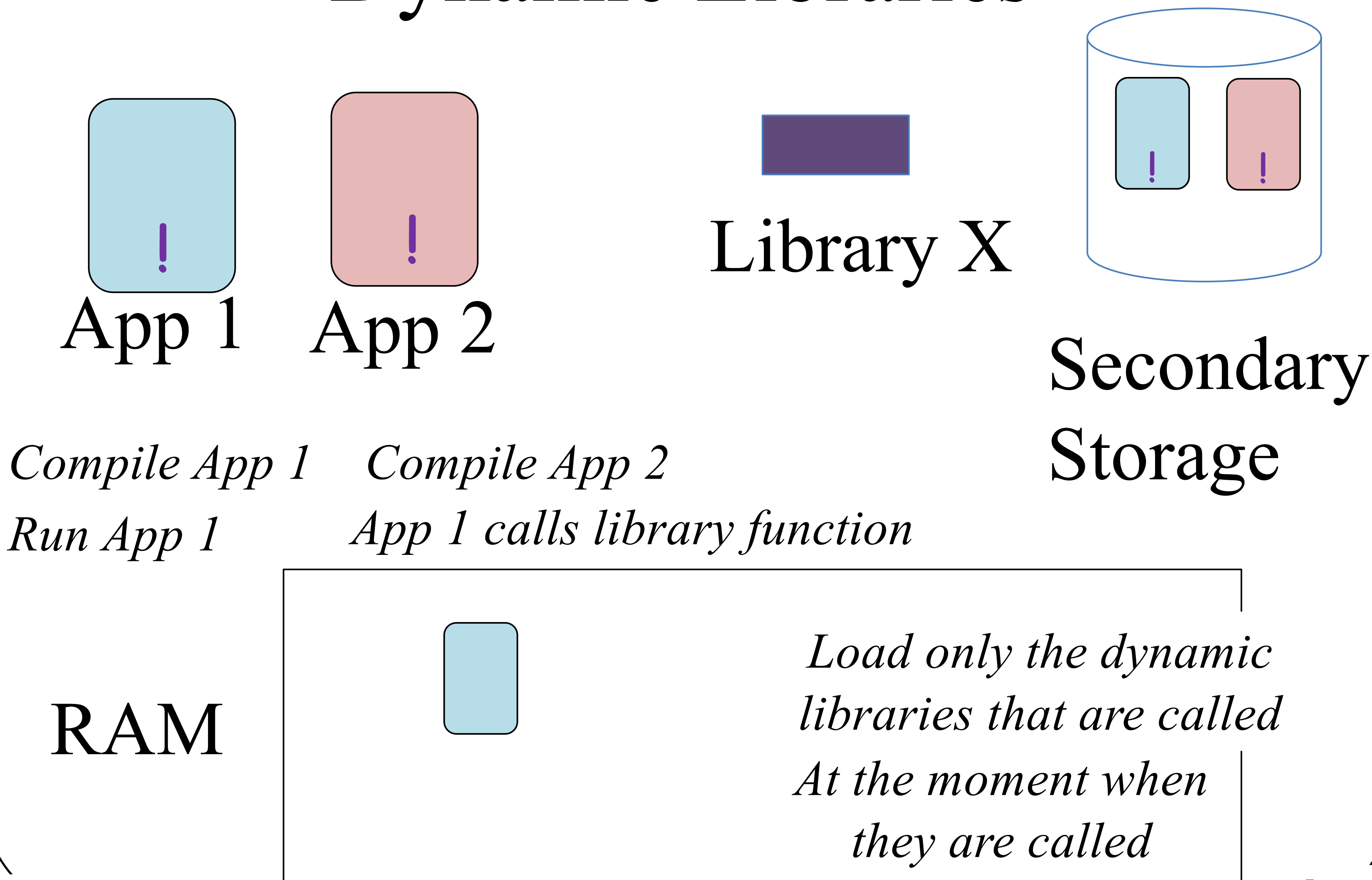
Compile App 1 *Compile App 2*

Run App 1 *Run App 2*

RAM



Dynamic Libraries



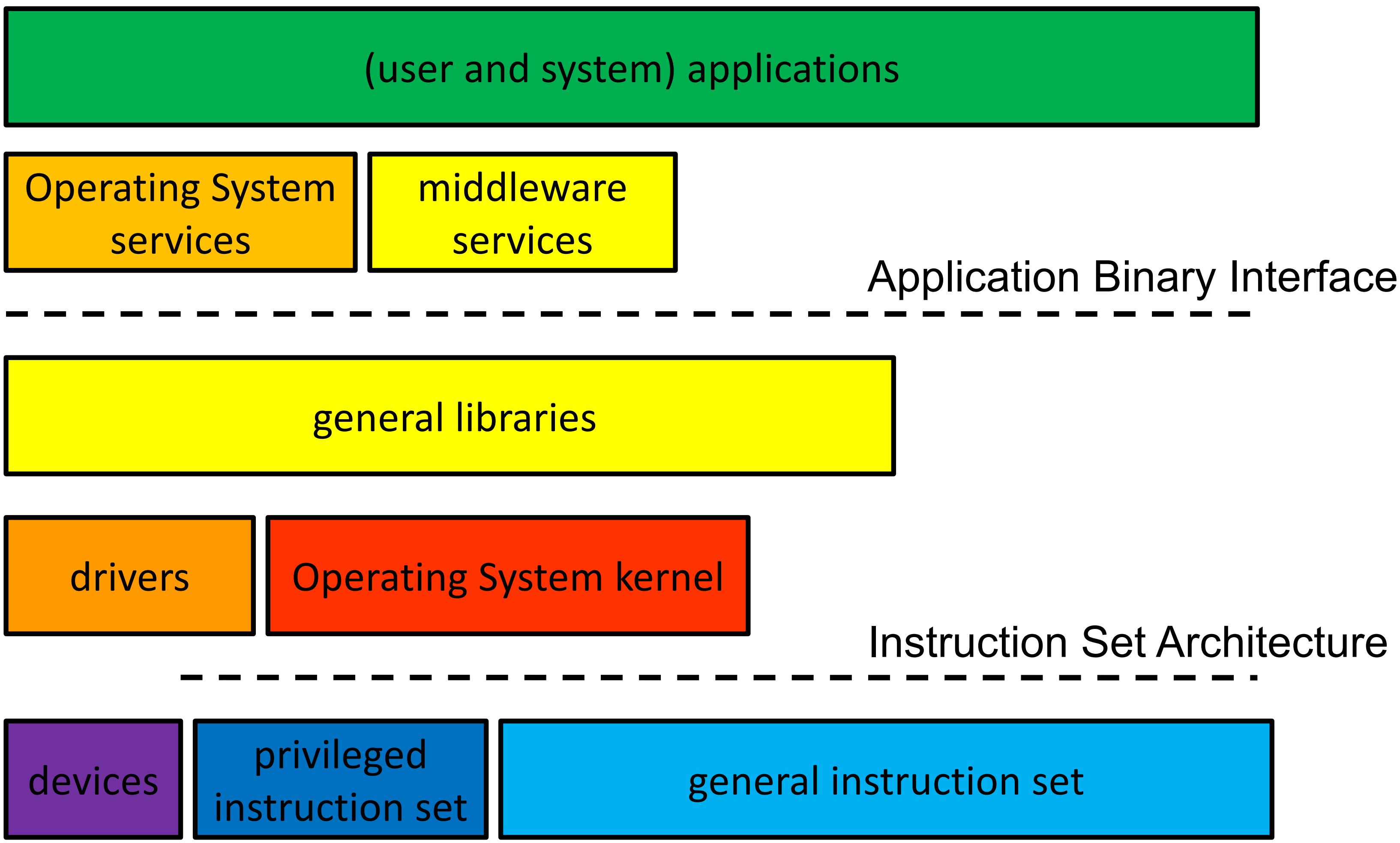
Service Delivery via System Calls

- 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

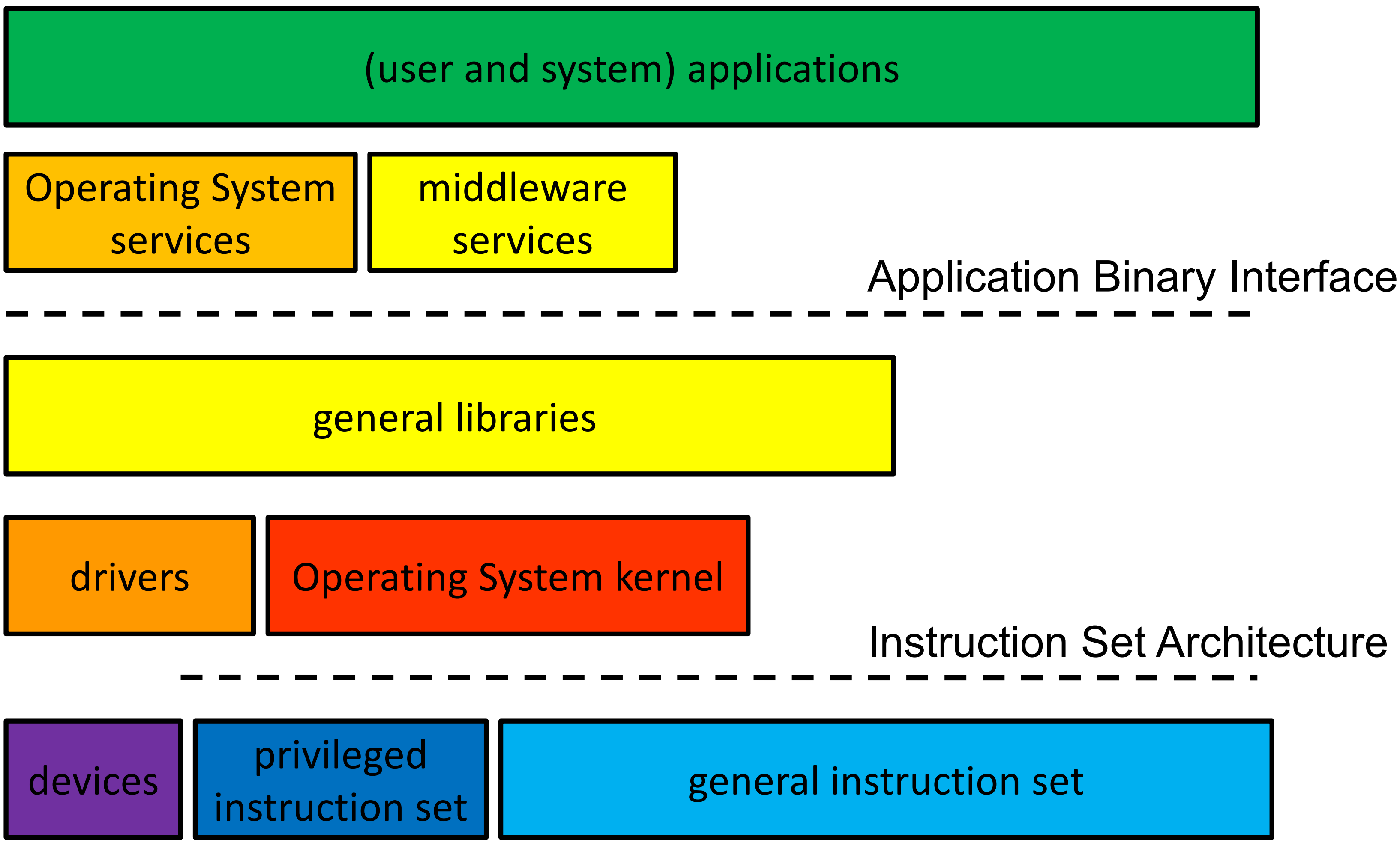
The Kernel Layer



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



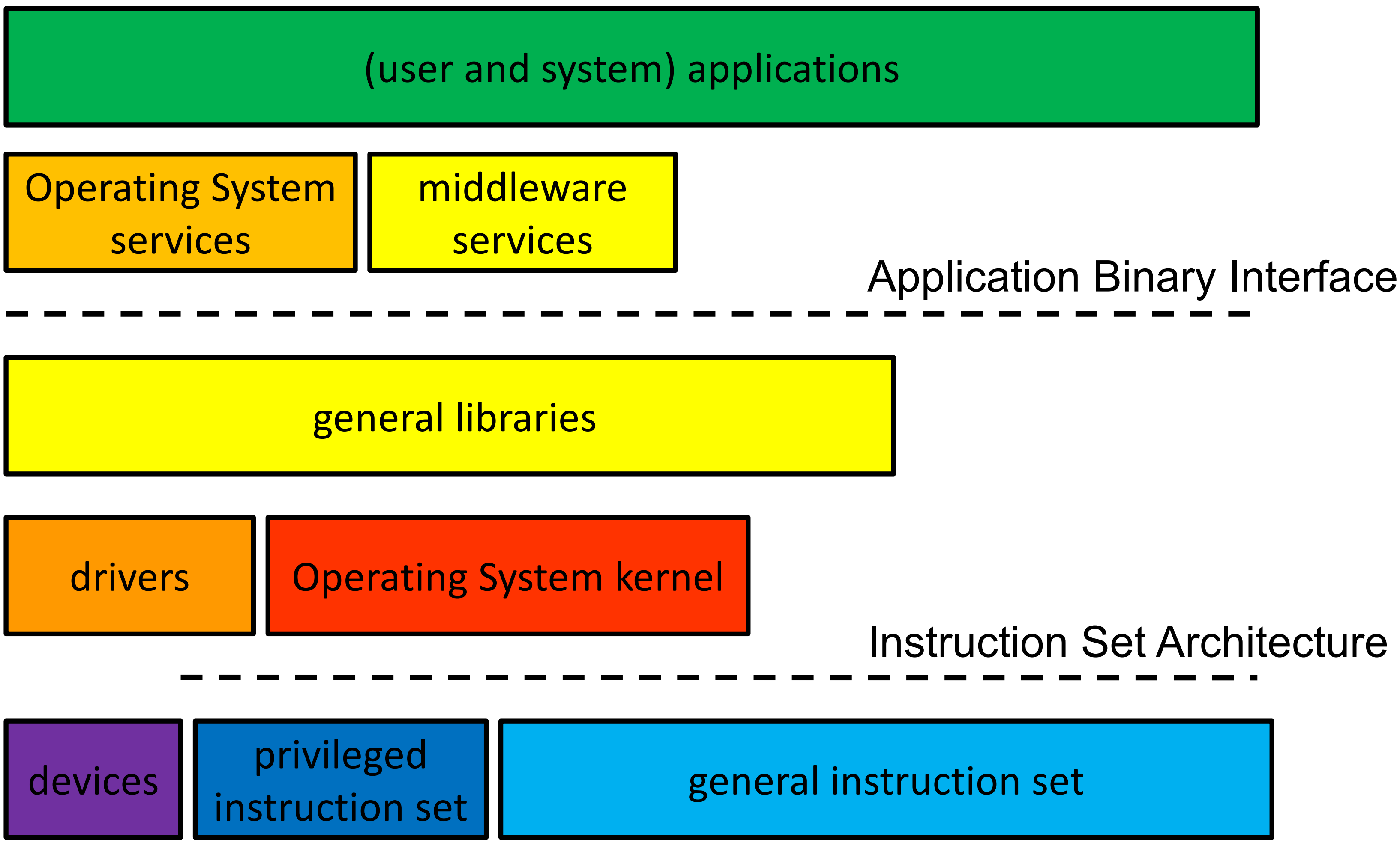
Service Delivery via Messages

- Exchange messages with a server (via syscalls)
  - Parameters in request, returns in response
- Advantages:
  - Server can be anywhere on earth (or local)
  - 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

System Services via Middleware

- Software that is a key part of the application or service platform, but <u>not part of the OS</u>
  - 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



Conclusion

- Operating systems have converged on a few popular systems
- Operating systems provide services via abstractions
- Operating systems offer services at several layers in the software stack