System Services

Abstraction

- Abstract resources as opposed to physical resources
    - Ex. File: abstraction or representation of physical data storage
    - Ex. Processes: abstraction of CPU and RAM processes

- Simpler and better suited for programmers and users
    - Easier to use than the original resources
    - Don't need to worry about the details of implementation
    - Ex. hide the slow erase cycle of flash memory (can't erase data), but it seems like we did delete data

- Convenient behavior
    - Ex. Make it look like you have a network interface entirely for your own use
    - Ex. Abstracting wifi and ethernet, which is handled by the OS

Generalizing Abstractions

- Many variations in machines' HW and SW
- Make many different types appear to be the same, since applications can deal with single common class
- Usually involves a common unifying model
    - Ex. PDF document format for printers
    - Printer drivers make different printers look the same
    - Browser plugins to handle multimedia data
- Abstractions cost time, but it's usually worth

OS Resources

- Serially Reusable Resources
    - Can be used by multiple clients, but can only be used by one application at any given time
        - Ex. Speakers, only one app can play music at a time
    - Requires access control to ensure exclusive use
    - Require graceful transitions from one user to the next
        - Ex. End audio on youtube and start on spotify
    - Ex. Sending network packets: If you download something while listening to music, packets of music and download occur in sequence (not at the same time)

- Partionable resources
    - Divide disjoin pieces for multiple clients, spatial multiplexing
    - Needs access control to ensure
        - Containment: you cannot access resources outside of you partition
        - Privacy: nobody else can access resources in your partition
        - Ex. Persistent storage device (users shouldnt be able to overwrite the OS with their own software (there are boundaries)

- Graceful transitions in partitions
  - Still important, most partitionable resources aren't permanently allocated
  - As long as you own the resource, then no one else can own it

- Shareable resources
  - Usable by multiple concurrent clients
  - Clients don't wait for access to resources and don't own a particular subset of the resource
  - May involve limitless resources (air in the room, or a copy of the OS shared by processes)

  - Graceful transitions in shareable resources aren't important since the resource doesn't
    - Shareable resource usually does not change state

OS Trends
- Larger and more sophisticated
- Changing roles:
  - OS moved to shielding applications from hardware and providing powerful apps with more computing power

- Core role of boundary between software and hardware remains
- OS best understood by capabilities added, application enabled, and problems they solve
- Applications have more complex internal behavior and complex interfaces, resulting in increased complications with the hardware

OS convergence
- Other than Windows, Mac, and Linux there are a few special OS for special purposes
- OS are expensive to build and hard to maintain (hard business to get into)
- Only succeed if users choose them over other options
- Need clear advantages over alternatives

OS services
- Offer services to other programs, generally as abstractions
- CPU / Memory
  - Processes, threads, VMs
  - Virtual address spaces, shared segments
- Persistent storage abstractions
  - Files and file systems
- Other I/O abstractions (windows, sockets, VPNs, signals)

Services: Higher Level Abstractions

- Cooperating parallel processes
    - Locks, condition variables

- Security
    - User authentication
    - Secure sessions, at rest encryption

- User Interface
    - Desktop, GUI widgets, management

Services: Lower level

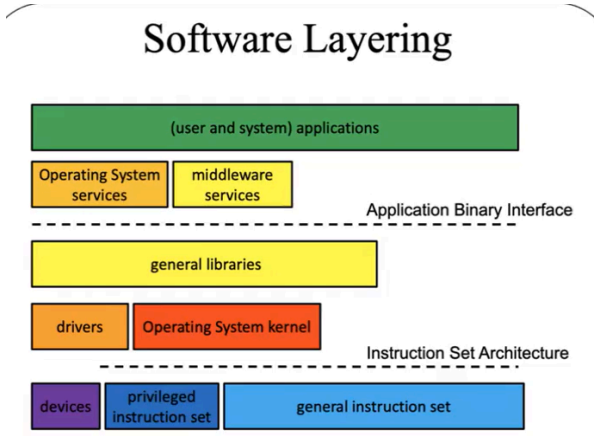
- Not visible to the user
- Enclosure management
    - Hot plug, power, fans
- Software updates and configuration registry
- Dynamic resource allocation and scheduling
- Networks, protocols, and domain services
    - USB, Bluetooth, TCP/IP

How OS delivers services

- Applications could 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 software stack

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
- Everything can be mapped down to hardware level



Service Delivery via Subroutines

- Function calls

- Access services via direct subroutine calls
  - Push parameters onto the stack, jump to subroutine, return values in registers
  - Typically done at high levels

- Advantages
  - Very fast: instructions are built into the hardware
  - Runtime binding: Don't have to know which function to call beforehand since binding occurs during execution

- Disadvantages
  - All services implemented in the same address space
  - Limited ability to combine different languages
  - Can't usually use privileged instructions (running at the application level, can only access standard instructions)

Service Delivery via Libraries

- Many subroutines that can be called
- One subroutine delivery approach
- Libraries are collection of object modules
  - Single file can contain many files
  - These modules can be used directly

Library Characteristics

- Reusable code makes it easier, single, well maintained copy
- Encapsulates complexity
- Simply code, no special privileges

Static

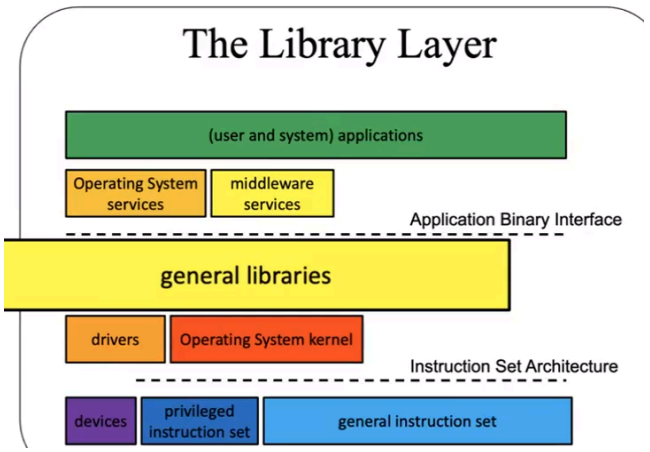
- Include in load module at link time

Shared

- Map into address space during execution

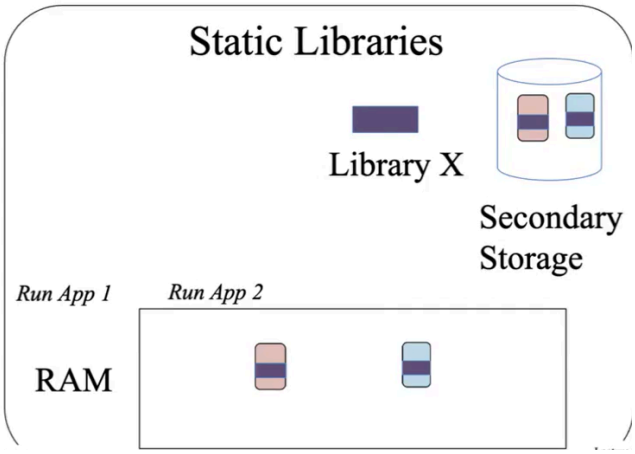
Dynamic

- Choose and load at run time



Static Libraries

- Added to a program's load module
- Each load module has its own copy of each library, increasing the size of each process
- Programs must be relinked to incorporate new library
- Existing load modules don't benefit from bug fixes since they have old code



- Compiling each app downloads the library code into the app
- Takes up space on persistent storage
- When we want to run both apps, both apps are moved into RAM.
- This means both apps with identical libraries are taking up valuable space on RAM

Shared Libraries

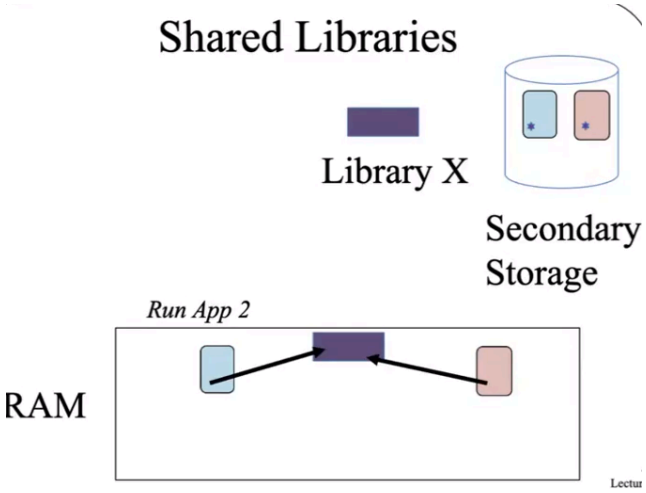
- On in memory copy, which is shared by all processes
- Keep the library separate from the load modules
- Operating system loads library along with program

  Advantages
  - Reduced memory consumption, since one copy shared by multiple processes

- Faster program startups: If the library is already in memory, it doesn't need to be loaded
- Simplified updates since the library isn't in the program load modules, so programs automatically get the newest version when restarted

Disadvantages

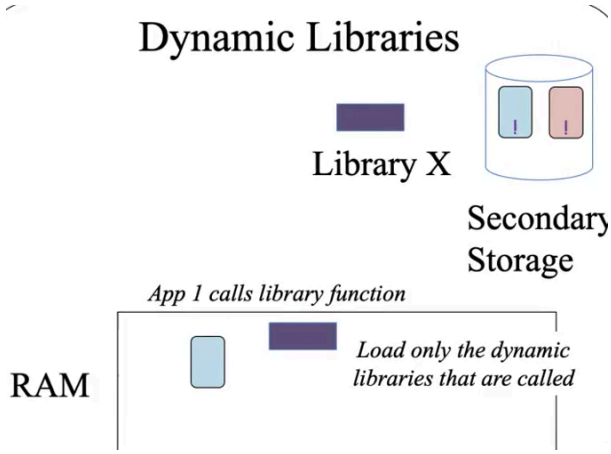
- Not all modules work in a shared library since they can't define global data storage
- Modules are added into program memory, even if they aren't needed
- Called routine must be known at compile time since symbols are



- When compiled, an app has a reference to the library
- When the app is run, the library is moved into RAM with the app
- This allows the apps to share the library

Dynamic Libraries

- Only loads libraries when it is used
- IF the code which needs the library is never run, then the library is never loaded



- Each app has a reference to the library
- Library is only loaded into RAM if the code which calls the library is run

Service Delivery via System Calls

- Switch from running app code to running OS code
- Generally we want processes to be self contained and not interfere with other processes, although the sharing of information is needed

- Forced entry into the operating system
  - Parameters and returns similar to subroutine
  - Implementation is in shared or trusted kernel

- Advantages
  - Able to allocate new or privileged resources (able to interact with hardware)
  - Able to share or communicate with other processes

- Disadvantages
  - Much slower than subroutine calls

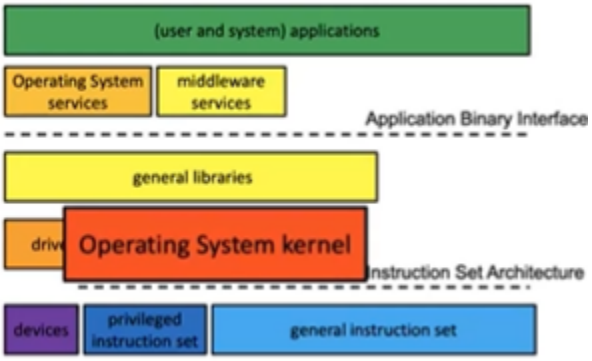
Kernel Services

- Calling library code (and the kernel) is the usual way to interact with the OS

- Functions which require privilege
  - Interrupts, I/O
  - Allocation of resources like memory
  - Ensuring process privacy and containment

- Some operations may be outsourced
  - System daemons, server processes
  - Daemons: programs that are always running, waiting for the system call to start it

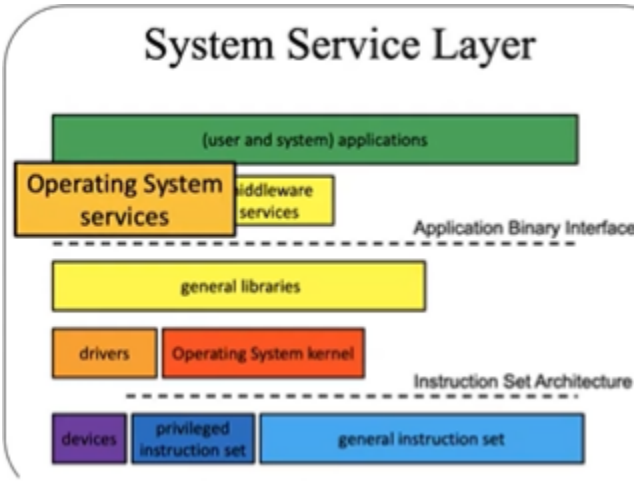
- Some plugins may be less trusted
  - Device drivers, file systems, network protocols



Outside Kernel Layer

- Not all trusted code must be in the kernel
    - May not need to access kernel data structures
    - May not need to execute privileged instructions

- Some are actually somewhat privileged process
    - Login can create / set user credentials



Messages System Delivery

- Exchange messages with a server
    - Parameters in request, returns in response

- Advantages:
    - Server can be anywhere on earth (or local)
    - Servers can be highly scalable and implemented in user-mode code

- Disadvantages:
    - Much slower than subroutine calls
    - Limited ability to operate on process resources

Middleware Services

- Special applications that receive and send messages

- Software that is a key part of the app, but not part of the OS
    - Database, messaging system
    - Hadoop, Openstack: cloud computing
    - Cassandra, Gluster: cloud data storages

- Kernel code is expensive and dangerous
    - User mode code is easy to build, test,and debug
    - User mode code is more portable
    - User mode code can crash and be restarted