Resources, Services & Interfaces

- 1C what is an Operating System
- 2A OS Services, Layers, and Mechanisms
- 2B Service Interfaces

What does Operating System do?

- manages the hardware
  - fairly allocate hardware among the applications
  - ensure privacy and enable controlled sharing
  - oversee program execution and handle errors
- abstract the bare hardware
  - easier to use, make apps platform-independent
  - provide private virtual computer for each program
- new abstractions to enable applications
  - powerful features beyond the bare hardware

What makes the OS special?

- It is always in control of the hardware
  - first software loaded when the machine boots
  - continues running while apps come and go
- Parts of it have complete access to hardware
  - privileged instructions, all memory and devices
  - mediates application access to the hardware
- It is trusted
  - to store, manage, and protect critical data
  - to perform all requested operations in good faith

Privileged Instructions

- most CPU instructions can be used by anyone
  - e.g. arithmetic, logical, data movement, flow control
- some instructions are privileged
  - e.g. operations associated with I/O, interrupts, virtual address spaces, and processor mode.
  - these could compromise data privacy or integrity
  - they can only be executed when in privileged modes
  - otherwise they are illegal operations (cause exception)
- the OS Kernel runs in privileged modes
  - giving it full control of the computer

What does an OS look like?

- applications see objects and operations
  - CPU supports data types and operations
    - bytes, shorts, longs, floats, pointers …
    - add, multiply, copy, compare, indirection, branch …
  - OS supports richer objects, higher operations
    - files, processes, threads, segments, ports, …
    - create, destroy, read, write, signal, …
- much of what OS does is behind-the-scenes
  - plug & play, power management, fault-handling, domain services, upgrade management, …

Services: Hardware Abstractions

- CPU/Memory abstractions
  - processes, threads, virtual machines
  - virtual address spaces, shared segments
  - signals (as execution exceptions)
- Persistent Storage abstractions
  - virtual LUNs, file systems, and files
  - databases, key/value stores, object stores
- other I/O abstractions
  - virtual terminal sessions, windows
  - sockets, pipes, VPNs, asynchronous events

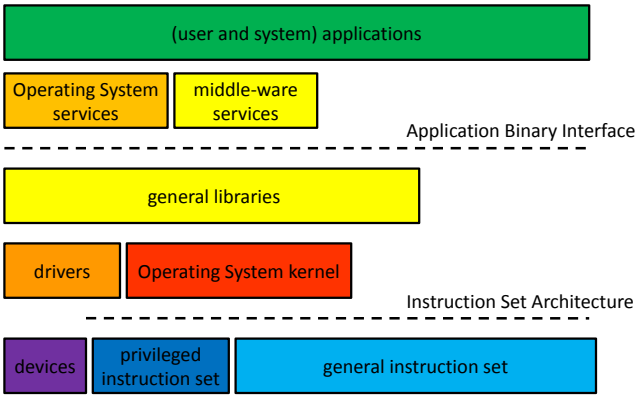
Services: Higher Level Abstractions

- cooperating parallel processes
  - locks, condition variables
  - distributed transactions, leases
  - distributed consensus, eventual consistency
- security
  - authenticated sessions, rule-based access control
  - secure sessions, at-rest (data) encryption
- external and user interfaces
  - hot-plug services, graphics, sound
  - GUI widgetry, desktop and window management

Services: under the covers

- low level resource management
  - power, fans, fault handling, device configuration
- 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



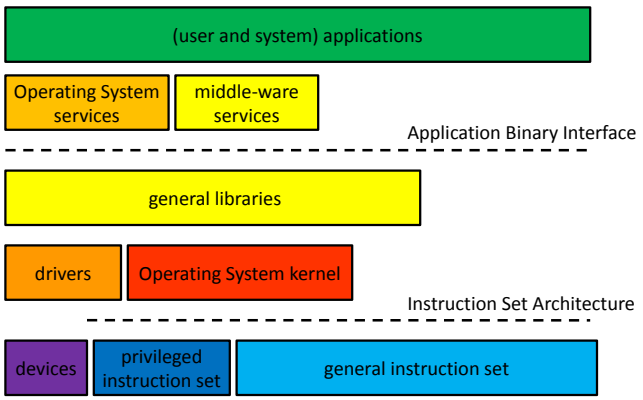
Service delivery via subroutines

- access services via direct subroutine calls
  - push parameters, jump to subroutine
  - return values in registers on on the stack
- advantages
  - extremely fast (nano-seconds)
  - implementation completely under our control
- disadvantages
  - all services implemented in same address space
    - can only access our process' resources
  - limited ability to combine different languages

Layers: libraries of subroutines

- convenient functions we use all the time
  - 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 … run-time binding, dynamic loading
- it is only code … no special privileges

Software Layering



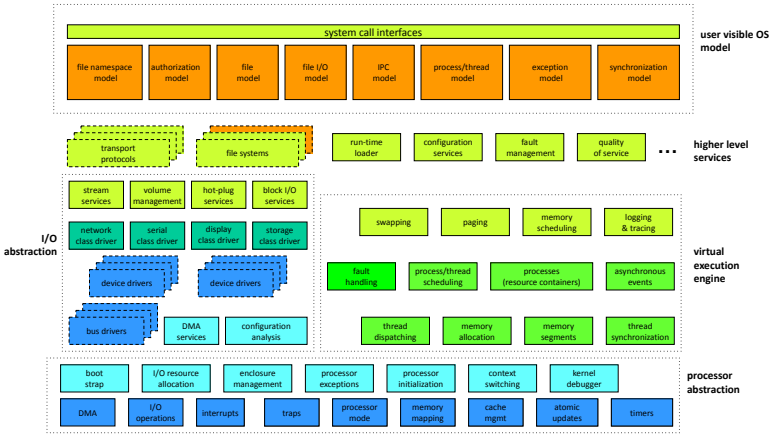
Service delivery via system calls

- force an entry into the OS kernel
  - 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
  - evolution is very slow and expensive

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-of-the-kernel
  - system daemons, server processes
- some plug-ins may be less-trusted
  - device drivers, file systems, network protocols

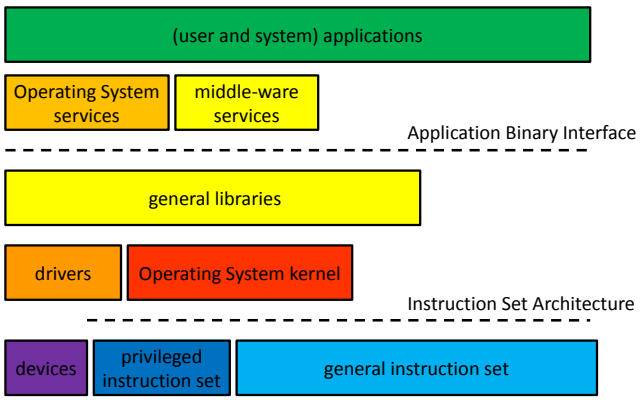
Kernel Structure (artists conception)



Service delivery via messages

- exchange messages with a server (via syscalls)
  - parameters in request, returns in response
- advantages:
  - distributed, client/server need not be co-located
  - 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

Software Layering



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 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 sender
  - NFS server is "*trusted*" to perform access control

Layers: middle-ware

- 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

Where to implement a service

- How many different applications use it?
- How frequently is it used?
- How performance critical are the operations?
- How stable/standard is the functionality?
- How complex is the implementation?
- Are there issues of privilege or trust?
- Is the service to be local or distributed?
- Are there to be competing implementations?

Is it faster if it is in the OS?

- OS is no faster than a user-mode process
  - CPU, instruction set, cache are the same
- some services involve expensive operations
  - servicing interrupts … faster in the kernel
  - making system calls … unnecessary in kernel
  - process switches … faster/less often in kernel
- but kernel code is very expensive
  - requires much more effort to build and test
  - complex, brittle, many rules

Why Do Interfaces Matter?

- primary value of OS is the apps it can run
  - it must provide all services those apps need
  - via the interfaces those apps expect
- correct application behavior depends on
  - OS correctly implements all required services
  - application uses services only in supported ways
- there are numerous apps and OS platforms
  - it is not possible to test all combinations
  - rather, we specify and test interface compliance

Application Programming Interfaces

- a source level interface, specifying
  - imports (e.g. Java/Python) and includes (e.g. C/C++)
  - data types, data structures, constants
  - macros, routines, parameters, return values
- a basis for software portability
  - recompile program for the desired ISA
  - linkage edit (or load) with OS-specific libraries
  - resulting program runs on that ISA and OS
- they are (by definition) language specific
  - but an API can be offered in many languages (per language interfaces to the same services)

Application Binary Interfaces

- a binary interface, specifying
  - load module, object module, library formats
    - what makes a blob of ones and zeroes a program
  - data formats (types, sizes, alignment, byte order)
  - calling sequences, linkage conventions
    - general (independent of particular routine semantics)
  - binding of an API to Instruction Set Architecture
- a basis for *binary compatibility*
  - one binary will run on any ABI compliant system
    - e.g. all x86 Linux/Gentoo/OS X/…
    - may even run on Windows

Other interoperability 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, WiFi, BlueTooth)
  - system management (e.g. DHCP, SNMP, LDAP)
  - remote data access (e.g. FTP, HTTP, CIFS, S3)

Reading and Assignments

Reading:

- Arpaci C3 … introduction to processes
- Arpaci C4 … processes
- Arpaci C5 … process APIs
- Arpaci C6 … process implementation

Projects:

- we will help w/project 0 problems in the lab

Supplementary Slides

What functionality is in kernel

- as much as necessary, as little as possible
  - this is very expensive to develop and maintain
- functionality must be in the OS if it …
  - required for security, trust, or resource integrity
- functionality must be in the kernel if it …
  - requires use of privileged instructions
  - requires manipulation of kernel data structures
- other simple functions can be in libraries
- complex functionality provided by services

Platforms

- ISA doesn't completely define a computer
  - functionality beyond user mode instructions
    - interrupt controllers, DMA controllers
    - memory management unit, I/O busses
    - BIOS, configuration, diagnostic features
    - multi-processor & interconnect support
  - I/O devices
    - display, disk, network, serial device controllers
- these variations are called "platforms"
  - the platform on which the OS must run

Portability to multiple ISAs

- successful OS will run on many ISAs
  - some customers cannot choose their ISA
  - if you don't support it, you can't sell to them
- minimal assumptions about specific h/w
  - general frameworks are h/w independent
    - file systems, protocols, processes, etc.
  - h/w assumptions isolated to specific modules
    - context switching, I/O, memory management
  - careful use of types
    - word length, sign extension, byte order, alignment

Binary Distribution Model

- binary is the opposite of source
  - a source distribution must be compiled
  - a binary distribution is ready to run
- one binary distribution per ISA
  - no need for special per-OEM OS versions
- binary model for platform support
  - device drivers can be added, after-market
    - can be written and distributed by 3rd parties
    - same driver works with many versions of OS

Binary Configuration Model

- eliminate manual/static configuration
  - enable one distribution to serve all users
  - improve both ease of use and performance
- automatic hardware discovery
  - self identifying busses
    - PCI, USB, PCMCIA, EISA, etc.
  - automatically find and load required drivers
- automatic resource allocation
  - eliminate fixed sized resource pools
  - dynamically (re)allocate resources on demand

Flexibility

- different customers have different needs
- we cannot anticipate all possible needs
- we must design for flexibility/extension
  - mechanism/policy separation
    - allow customers to override default policies
    - changing policies w/o having to change the OS
  - dynamically loadable features
    - allow new features to be added, after market
    - file systems, protocols, load module formats, etc.
  - feature independence and orthogonality

Interface Stability

- people want new releases of an OS
  - new features, bug fixes, enhancements
- people also fear new releases of an OS
  - OS changes can break old applications
- how can we prevent such problems?
  - define well specified Application Interfaces
  - apps only use committed interfaces
  - OS vendors preserve upwards-compatibility