Lecture Notes (Before Class)

Services: Hardware Abstractions

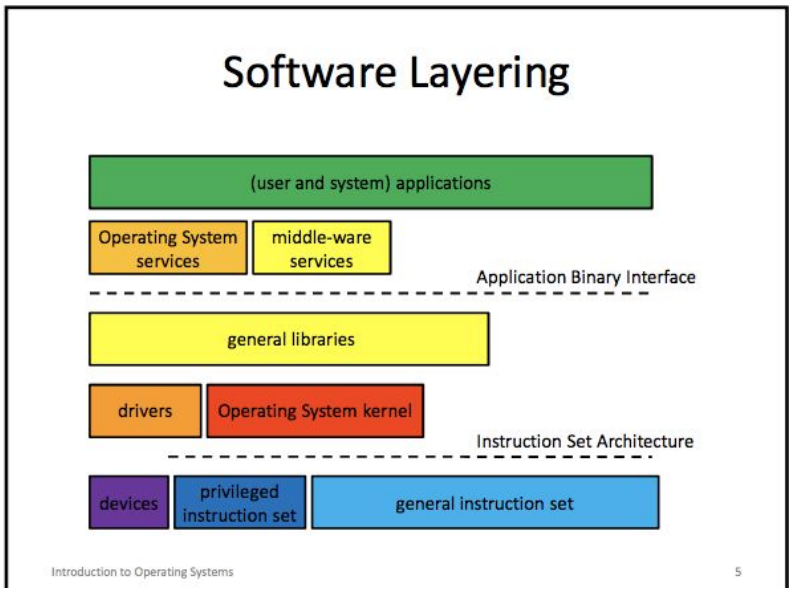
- CPU/Memory Abstractions
    - processes (a virtual CPU), threads, virtual machines
        - processes are basically virtual CPUs
        - threads were implemented after people decided that processes were too expensive
    - virtual address spaces, shared segments
        - virtual address space is analogous to memory
    - signals (as execution exceptions)
    - CPU: executes instructions
- Persistent Storage Abstractions
    - files and file systems, virtual LUNs
        - files do not resemble disks
    - databases, key/value stores, object stores
- Other I/O abstractions
    - virtual terminal sessions, windows
    - sockets, pipes, VPNs, signals (as interrupts)

Services: Higher Level Abstractions

- Cooperating parallel processes
    - locks, condition variables
        - you need locks to save personal data so that others cannot access them
        - locks: can create problems such as race conditions
    - distributed transactions, leases
        - solutions to control the chaos of locks
- Security
    - user authentication
    - secure sessions, at-rest encryption
- User Interface
    - GUI widgetry, desktop and window management
    - multi-media

Services: under the covers

- 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



*Application Binary Interface: Everything you need to know in terms on ones and zeroes
Definition: The application binary interface is the binding of an application programming interface to an instruction set architecture (ISA of computer)

- The ones and zeroes of different machines may be different

Service Delivery via Subroutines (a bunch of code that was, in principle, separately compiled/packaged)

- Access services via direct subroutine calls
    - push parameters, jump to subroutine, return values in registers on on the stack
- advantages
    - extremely fast (nanoseconds)
    - DLLs enable run-time implementation binding
    - dynamically loadable libraries (DLLs): choose a library to load on runtime
- disadvantages
    - all services implemented in same address space
    - limited ability to combine different languages
    - limited ability to change library functionality

Layers: libraries (a library can be considered as a collection of subroutines)

- convenient functions we use all the time
    - reusable code makes programming easier
    - a single well written/maintained copy
    - encapsulates complexity
- 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**
    - **bindings**

        - **static:** stuck with the ones and zeroes for the rest of your program (permanent) A statically linked library generally means that it is pulled into the load module shortly after you compile it
        - exec: the operation of starting a new program in a process. If there's a new version of a library, if you stop your program and rerun it, it will choose the new library. (Load the library into the process)
        - run: up until you reference that object, the program may not know that the library exists
- It is only code …. it has no special privileges

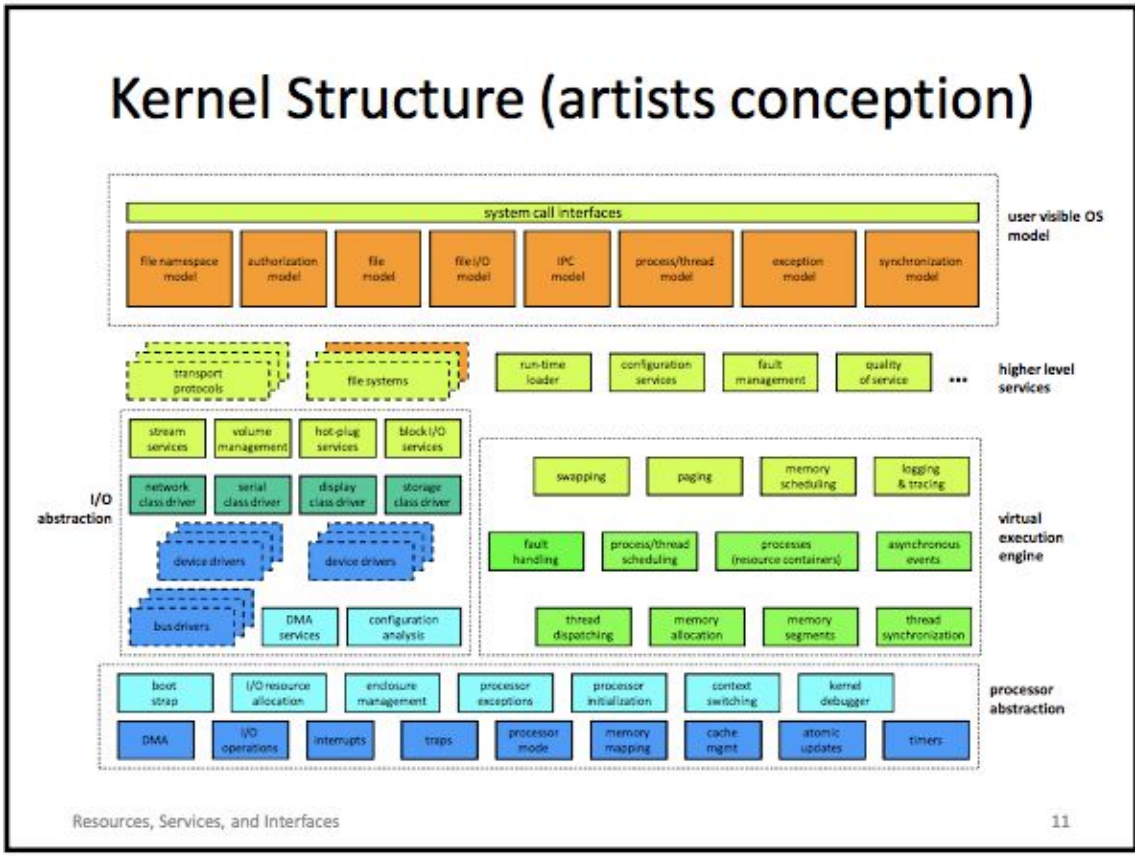
***DIFFERENCES BETWEEN DYNAMIC AND SHARED LIBRARY WILL BE ON MIDTERM (ADVANTAGES/DISADVANTAGES)**

Service delivery via system calls

- forces 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 the subroutine calls (at the basic level, with no complexities)
    - 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-sourced
    - system daemons, server processes
- some plug-ins may be less trusted
    - device drivers, file systems, network protocols
- Kernel: the minimal set of functions you need to implement
- ex: In Ubuntu, the kernel is the stuff that is loaded up into the computer, before tha applications is run. (Includes the stuff in the picture below)



Service delivery via messages

- A service is a thing of value that somebody else does for you. A subroutine is a mechanism for requesting a service)
- 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
- disadvantages
    - 1000x-10000x slower than subroutines
    - limited ability to operate on process resources

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 labeled messages
    - NFS server is trusted to honor access control data

Layers: middle-ware

- Something that is not part of the OS but everyone is using it. (ex: published subscribed frameworks)
- platform: an instruction set architecture. In a more general sense, is all the stuff that my application expects to find available on the computer
- A lot of the middle-ware becomes the platform of many applications
- software that is a key part of the application or service platform, but NOT part of the OS
    - database, pub/sub messaging system
    - Apache (web server), 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 is much more more portable
    - user-mode code can crash and be restarted

Where to implement a service

- How many different apps 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 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, difficult to test and build, and difficult to change

Why do Interfaces Matter?

*just because one program works on a UNIX system it doesn't mean that it'll work on another because it may not be a part of the standard

*We must give interfaces in great detail

- 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 implementing all required services
    - Application using 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

APIs

- a source level interface specifying include files, data, types, data structures, constants, macros, routines, parameters, and return values
- a basis for software portability
    - recompile program for desired ISA
    - linkage edit with OS-specific libraries
        - ***Linkage editing and load processing programs prepare the output of language translators for executions
        - linker: links object files into a single executable
    - resulting binary runs on that ISA and OS
- they are, by definition, language specific

ABIs

- 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
        - it is independent of particular routine semantics
- A basis for binary compatibility
    - one binary will run on any ABI compliant system
        - e.g. all x86 Linux/BSD/OSx/Solaris
        - may even run on windows platforms!
- ABI does not include the ISA: The ABI is the binding of an API to a particular ISA
- Think about how we need to translate APIs to ones and zeros
- x86 is an ISA that does not implement any system calls, the ABI defines the system calls

Other interoperability interfaces

- Data formats and information encodings
    - multi-media content (e.g. MPS, 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
    - remote data access

Interoperability requires completeness

- Interface specification must be complete

    - all input parameter, all output values, all input/output behaviors, all internal states and errors
- All specifications should be explicit
    - no undocumented features
    - not defined by an implementation's behavior

Interoperability also 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
- Rather, 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

Interoperability requires stability

- No program is an island
    - programs use system calls, library routines, operate on external files, exhange messages with other software
- 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

Compatibility Taxonomy

- upwards compatible
    - new version will support older interfaces
        - "i can talk to this and newer"
        - Remember Java applications: versioned interface
- backwards compatible
    - will correctly interact with old protocol versions
        - "i can talk to old people too"
- versioned interface, version negotiation
    - parties negotiate a mutually acceptable version
- compatibility layer
    - a cross-version translator
- non-disruptive upgrade
    - update components one-at-a-time w/o down-time
    - it should mean that the new one should support the old one without rebooting the operating system

Services: 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

Better Objects and Operations

- easier to use than the original resources
    - disk I/O without DMA, interrupts and errors
- compartmentalize/encapsulate complexity
    - a securely authenticated/encrypted channel
    - the message is private and secure and nobody will see it
    - create a new abstraction that is easy for people to use
- eliminate behavior that is irrelevant to user
    - hide the slow erase cycle of flash memory
- create more convenient behavior
    - highly reliable/available storage from anywhere

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

Generalizing Abstractions

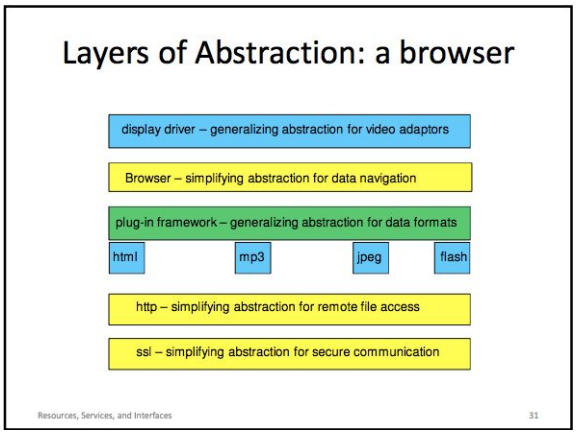
*Creating a common mechanism

- 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
    - device-specific drivers
    - browser plug-ins to handle multi-media data

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



Building Blocks and World Views

- An OS is a general purpose platform
  - it must support a wide range of applications
  - including those to be designed in the future
- OS services are software building blocks
  - not solutions, but pieces for building solutions
- OS abstractions represent a world view
  - concepts that encompass all possible s/w
  - interaction rules to govern their combinations
  - frame (guide/constrain) all future discussions

Virtualizing Physical Resources

- serially reusable (temporal multiplexing)
  - used by multiple clients, one at a time
  - requires access control to ensure exclusive access
- partitionable resources (spatial multiplexing)
  - different clients use different parts at same time
  - requires access control for containment/privacy
- sharable (no apparent partitioning or turns)
  - often involves mediated access
  - often involves under the covers multiplexing

Serially Reusable Resources

- Used by multiple clients … one at a time
  - temporal multiplexing
- Require access control to ensure exclusivity
  - while A has resource, nobody else can use it
- Require graceful transitions between users
  - B will not start until A finishes
  - B receives resource in like-new condition
- Examples printers, bathroom stalls

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: RAM, hotel rooms

Shareable Resources

- Usable by multiple concurrent clients
    - Clients do not have to wait for access to resources
    - Clients don't own a particular subset of resource
- May involve limitless resources
    - Air in a room, shared by occupants
    - Copy of the operating system, shared by all processes
- May involve under the cover multiplexing
    - cell phone channel (time and frequency multiplexed)
    - shared network interface (time multiplexed)

Interface Stability

Interface Specifications

*If each part is manufactured and measured to be within its specifications, then any collection of these parts can be assembled into a working whole. This greatly reduces the cost and difficulty of building a working system out of component parts

- The above principle applies to software as well. If you want to open a file on a Posix system, you use the open system call. There may be hundreds of different implementations of open but this doesn't matter because they conform to the same specification
    - Your work as a programmer is simplified, because you don't have to write your own I/O routines
    - Your program is likely to be easily portable to any Posix compliant system
    - Training time for new programmers is reduced, because everybody already knows how to use Posix I/O functions

The Criticality of Interfaces in Architecture

- A system architecture defines the major components of the system, the functionality of each, and the interfaces between them. Therefore, it is obvious that interfaces are going to be important in architecture
- It should be possible to independently implement and design interfaces. In principle, any implementations that satisfy those functional and interface specifications should combine to yield a working system

The Importance of Interface Stability

- We write contracts so that everybody knows what will be expected of them, and what they can expect from the other parties
- A software interface specification is a form of a contract
    - the contract describes an interface and the associated functionality
    - implementation providers agree that their systems will conform to the specification
    - developers agree to limit their use of the functionality to what is described in the interface spec

Interfaces vs. Implementations

- Suppose that someone writes a handy library to efficiently provide reliable communication over an unreliable network
    - I download their dev kit and start using it, and a few weeks into the project I realize that I need a feature that is not included in their documentation, but after reading their code, I discover that the needed feature is actually available
    - Six months later, they release a new version of their reliable library, and my product immediately breaks. After debugging, I discover that they have changed the undocumented code that I was depending on.
    - It turns out that I had not designed my product to work with their **interfaces**. My product only worked with a particular **implementation…** and implementations change.
    - implementation is a particular use of a product while their interface is what is the function
- An interface should exist independently from any particular implementation.

Program vs. User Interfaces

- Human beings are amazingly robust. We could change the text in dialog boxes, rearrange input forms, add new required input items, etc. This inclines many developers to be cavalier in making changes to the user interface
- However, code is nowhere nearly so robust. If I expect my second parameter to be a file name, and you pass me the address of a call-back routine instead, the best we can hope for is a good error message and a quick core-dump with no data corruption
- If you are going to be delivering binary software to non-developers, you have to trust that whatever platform they run it on will correctly implement all of the interfaces on which your program depends
- If components exchange services, and I make an incompatible change to the interfaces of one component, this has the potential to break other components in the same system. I can fix the other components in our system to work with the new changes bust:
    - this complicates the making of the change
    - we have to ensure that mismatched component sets are impossible

Is every interface carved in stone?
Absolutely not

- You are free to add features that do not change the interface application. In many cases you can add upwards-compatible extensions (all old programs will still work the same way, but new interfaces enable new software to access new functionality). There are a few ways to add incompatible changes to old interfaces without breaking backwards compatibility.
- Interface Polymorphism
    - In old school languages, a routine had a single interface specification. But more contemporary programming languages support polymorphism (different method signatures with different parameter and return types). If different versions of a method are readily distinguishable, it may be possible to provide new interfaces to meet new requirements, while continuing to support the older interfaces for backwards compatibility.
- Versioned Interfaces
    - backwards compatibility requirements do not prevent us from changing interfaces; they merely require us to continue providing old interfaces to old clients. In many cases, it may be possible for an application to call out which version of an interface it requires
    - How stable an interface needs to be depends on how you intend to use it. What is important is that
        - we be honest about our intentions and set realistic expectations
        - we have a plan for how we will manage change

Interface Stability and Design

- If we want to expose an interface to unbundled software with high quality requirements, we are not allowed to change it in non-upwards-compatible ways.
- When we design a system, and the interfaces between the independent components, we need to consider all of the different types of change that are likely going to happen

Software Interface Standards

- If applications are to readily ported to all platforms, all platforms must support the same services. This means that we need detailed specifications for all services, and comprehensive compliance testing to ensure the correctness of these implementations

Challenges of Software Interface Standardization

- When a technology or service achieves wide penetration, it becomes important to establish standards. ex: television would have never happened if each broadcaster and manufacturer had chosen different formats for TV signals.
- Standardization is a double edged sword: it is good so that nothing is heavily favored over another, creating disadvantages for different providers. They also tend to have very clear and complete specifications and well developed conformance testing procedures. They also give technology suppliers considerable freedom to explore alternative implementations (e.g. to improve reliability, performance, etc.)

    - However, they greatly constrain the range of possible implementations. An interface will specify who provides what information. If a new interface is not 100% compatible with other products and old ones, the improved design can be non-compliant
    - Interface standards also impose constraints on their consumers. An application that has been written to a well-supported standard can have high confidence of working on any compliant platform.
    - It becomes difficult to change standards if many people depend on them as well.

Confusing Interfaces with Implementation

- Interface standards should not specify design. Rather they should specify behavior, in as implementation-neutral a way as possible
- But true implementation neutrality is also very difficult to achieve because existing implementations often provide the context against which we frame our problem statements.
- It is common to find that a standard developed ten years ago cannot reasonably be extended to embrace a new technology because it was written around a particular set of implementations
- Reverse engineering can be detrimental
    - Since the implementation was not developed to or tested against the interface specification, it may not actually comply with it
    - In the absence of specifications, it may be difficult to determine whether particular behavior is a fundamental part of the interface or a peculiarity of the current implementation
    - “if it isn’t in writing, it doesn’t exist”

The rate of evolution, for both technology and applications

- The technologies that go into computers are evolving quickly and our software must evolve to exploit them. The types of applications we develop are also evolving dramatically.
- Maintaining stable interfaces in the face of fast and dramatic evolutions is extremely difficult.
    - ex: Microsoft Windows was designed as a single-user Personal Computer operating system - making it very difficult to extend its OS service APIs to embrace the networked applications and large servers that dominate today’s world.
    - When faced with such change we find ourselves forced to choose between
        - Maintaining strict compatibility with old interfaces, and not supporting new applications and/or platforms.
        - Developing new interfaces that embrace new technologies and uses, but are incompatible with older interfaces and applications
        - A compromise that partially supports new technologies and applications, while remaining mostly compatible with old interfaces

Proprietary vs Open Standards

- A Proprietary interface is one that is developed and controlled by a single organization (ex: the MIcrosoft Windows API).
- An Open Standard is one that is developed and controlled by a consortium of providers and/or consumers.
- Whenever a technology provider develops a new technology they must make a decision based on:
    - Should they open their interface definitions to competitors, to achieve a better standard?
        - Reduced freedom to adjust interfaces to respond to conflicting requirements
        - Giving up the competitive advantage that might come from being the first/only provider
        - Being forced to re-engineer existing implementations to bring them into compliance with committee-adopted interfaces
    - Should they keep their interfaces proprietary, perhaps under patent protection?
        - If a competing, open standard evolves, and ours is not clearly superior, it will eventually lose and our market position will suffer as a result
        - Competing standards fragment the market and reduce adoption
        - With no partners, we will have to shoulder the full costs of development and evangelization. However, with an open standard, these costs can be distributed over a wider population

Application Programming Interfaces (APIs)

- Definition: A system of tools and resources in an OS that enables developers to create applications
- Most of the time, when we look up and exploit some service, we are working with APIs. A typical API specification is open(2) which includes
    - A list of included routines/methods, and their signatures (parameters, return types)
    - A list of associated macros, data types, and data structures
    - A discussion of the semantics of each operation, and how it should be used
    - A discussion of all of the options, what each does, and how it should be used
    - A discussion of return values and possible errors
    - The specifications may also refer to sample usage scenarios as well
- The most important thing to understand about API specifications is that they are written at the source programming level. They describe how source code should be written in order to exploit these features. API specifications are a basis for software portability.
    - the benefit for application developers is that an application written to a particular API should easily recompile and correctly execute on any platform that supports that API
    - The benefit for the platform suppliers is that any application written to supported APIs should easily port to their platform

- Sometimes, this promise can only be delivered if the API has been defined in platform-independent ways
    - ex: An API defined some type as int (implicitly assuming that to be at least 64 bits wide) might not work on a 32 bit machine
    - ex: An API that accessed individual bytes within an int might not work on a big endian machine

Application Binary Interfaces (ABIs)

- Well-defined APIs are not always enough
- Most users do not have access to the sources for most of the software they use, and probably couldn't successfully compile it if they did. Most people just want to download a program and run it
- But (ignoring interpreted languages like Java and Python) executable programs are compiled (and linkage edited) for a particular instruction set architecture and operating system.
- How is it possible to build a binary application that will (with high confidence) run on any Android phone, or any x86 Linux?
- ***An Application Binary Interface is the binding of an API to an Instruction Set Architecture (ISA)
    - An ISA is a set of instructions supported by a computer: ex what bit patterns correspond to what operations
- An API defines subroutines, what they do, and how to use them.
    - An ABI describes the machine language instructions and conventions that are used to call routines. (How compilers build the application, like who cleans the stack that or how parameters are passed to functions)
    - A typical ABI contains things like
        - The binary representation of key data types
        - The instructions to call to and return from a subroutine
        - Stack-frame structure and respective responsibilities of the caller and callee
        - How parameters are passed and return values are exchanged
        - Register conventions
        - The analogous conventions for system calls, and signal deliveries
        - The formats of load modules, shared objects, and dynamically loaded libraries
    - The portability benefits of an ABI are much greater than those for an API. If an application is written to a supported API, and compiled and linkage edited by ABI-compliant tools, the resulting binary load module would correctly run.
    - As long as the CPU and Operating System support an ABI, there should be no need to recompile a program to be able to run it on a different CPU, OS, distribution, or release

Who actually uses the ABIs

- Most programmers will never directly deal with an ABI, as it is primarily used by
    - the compiler
    - the linkage editor, to create load modules
    - the program loader, to read load modules into memory
    - the operating system to process system calls