Outline

• Important properties for an operating system
• Critical abstractions for operating systems
• System services
Important OS Properties

• For real operating systems built and used by real people
• What’s most important depends on who you are talking about
  – Users
  – Service providers
  – Application developers
  – OS developers
• All are important clients for operating systems
For the End Users,

• Reliability
• Performance
• Upwards compatibility in releases
• Support for differing hardware
  – Currently available platforms
  – What’s available in the future
• Availability of key applications
• Security
Reliability

• Your OS really should never crash
  – Since it takes everything else down with it
• But also need dependability in a different sense
  – The OS must be depended on to behave as it’s specified
  – Nobody wants surprises from their operating system
  – Since the OS controls everything, unexpected behavior could be arbitrarily bad
Performance

• A loose goal
• The OS must perform well in critical situations
• But optimizing the performance of all OS operations not always critical
• Nothing can take too long
• But if something is “fast enough,” adding complexity to make it faster not worthwhile
  – Often overlooked by OS researchers and developers
Upward Compatibility

• 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
• What makes the compatibility issue manageable?
  – Stable interfaces
Stable Interfaces

• Designers should start with well specified Application Interfaces
  – Must keep them stable from release to release

• Application developers should only use committed interfaces
  – Don’t use undocumented features or erroneous side effects
Interfaces and Standards

- Standards in the Dark Ages (1965)
- The S/W Reformation (1985)
- The role of standards today
- APIs
- ABIs
Standards in the Dark Ages (1965)

- No software industry as we now know it
- All the money was made on hardware
  - But hardware is useless without software
  - All software built by hardware suppliers
  - Platforms were distinguished by software
- Software portability was an anti-goal
  - Keep customers captive to your hardware
  - Portability means they could go elsewhere
- Standards were few and weak
The S/W Reformation (1985)

• An outgrowth of the popular commodity PC
• The advent of the “killer application”
  – Desk-top publishing, spreadsheets, ...
  – The rise of the Independent Software Vendor
• Fundamental changes to platform industry
  – The “applications, demand, volume” cycle
  – Application capture became strategic
• Applications portability also became strategic
  – Standards are the key to portability
  – Standards compliance became strategic
Standards Today

• There are many software standards
  – Subroutines, protocols and data formats, …
  – Both portability and interoperability
  – Some are general (e.g. POSIX 1003, TCP/IP)
  – Some are very domain specific (e.g. MPEG2)

• Key standards are widely required
  – Non-compliance reduces application capture
  – Non-compliance raises price to customers
  – Proprietary extensions are usually ignored
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
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 for users
For the Service Providers,

- Reliability
- Performance
- Upwards compatibility in releases
- Platform support (wide range of platforms)
- Manageability
- Total cost of ownership
- Support (updates and bug fixes)
- Flexibility (in configurations and applications)
- Security
For the Application Developers,

- Reliability
- Performance
- Upwards compatibility in releases
- Standards conformance
- Functionality (current and roadmap)
- Middleware and tools
- Documentation
- Support (how to ...)

For the OS Developers,

• Reliability
• Performance
• Maintainability
• Low cost of development
  – Original and ongoing
Maintainability

• Operating systems have very long lives
  – Solaris, the “new kid on the block,” came out in 1993
  – Even smart phone OSes have roots in the 80s or 90s

• Basic requirements will change many times

• Support costs will dwarf initial development

• This makes maintainability critical

• Aspects of maintainability:
  – Understandability
  – Modularity/modifiability
  – Testability
Maintainability: Understandability

- Code must be learnable by mortals
  - It will not be maintained by the original developers
  - New people must be able to come up to speed
- Code must be well organized
  - Nobody can understand 1 million lines of random code
  - It must have understandable, hierarchical structure
- Documentation
  - High level structure, and organizing principles
  - Functionality, design, and rationale for modules
  - How to solve common problems
Why a Hierarchical Structure?

• Not absolutely necessary, but . . .
• Hierarchical layers usually understandable without completely understanding the implementation
• Expansion of one sub-system in a hierarchy usually understandable without understanding the expansion of other sub-systems
• Other structures tend not to have those advantages
Maintainability: Modularity and Modifiability

• Modules must be understandable in isolation
  – Modules should perform coherent functions
  – Well-specified interfaces for each module
  – Implementation details hidden within module
  – Inter-module dependencies should be few/simple/clean

• Modules must be independently changeable
  – Lots of side effects mean lots of bugs
  – Changes to one module should not affect others

• Keep It Simple Stupid
  – Costs of complexity usually outweigh the rewards
Side Effects

• A side effect is a situation where an action in one object has non-obvious consequences
  – Perhaps even to other objects
  – Generally not following the interface specification

• Side effects often happen when state is shared between seemingly independent modules and functions

• Side effects lead to unexpected behaviors

• And the resulting bugs can be hard to find
Maintainability: Testability

• OS must work, so its developers must test it
• Thorough testing is key to reliability
  – All modules must be thoroughly testable
  – Most modules should be testable in isolation
• Testability must be designed in from the start
  – Observability of internal state
  – Triggerability of all operations and situations
  – Isolability of functionality
• Testing must be automated
  – Functionality, regression, performance,
  – Stress testing, error handling handling
Automated Testing

• Why is it important that testing be automated?
• Automated tests can be run often (e.g. after every change) with very little cost or effort
• Automatically executed tests are much more likely to be run completely and correctly every time
• And discrepancies are much more likely to be noted and reported
Cost of Development

• Another area where simplicity wins
• If it’s simple, it will be quicker and cheaper to build
• Even better, there will be fewer bugs
  – And thus less cost for bug fixes
• And changing/extending it will be cheaper
• Low cost development usually implies speedy development
  – Quicker time to market
Critical OS Abstractions

• One of the main roles of an operating system is to provide abstract services
  – Services that are easier for programs and users to work with
• What are the important abstractions an OS provides?
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
The Basic Memory Operations

• Regardless of level or type, memory abstractions support a couple of operations
  – WRITE(name, value)
    • Put a value into a memory location specified by name
  – value <- READ(name)
    • Get a value out of a memory location specified by name

• Seems pretty simple

• But going from a nice abstraction to a physical implementation can be complex
Some Complicating Factors

• Persistent vs. transient memory
• Size of 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?

• At the bottom, the OS doesn’t have abstract devices with arbitrary properties

• It has particular physical devices
  – With unchangeable, often inconvenient, properties

• The core OS abstraction problem:
  – Creating the abstract device with the desirable properties from the physical device without 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
• If there are several reads/writes to the file, we expect each to occur in some order
  – With respect to the others
What Is Implementing the File?

• Most commonly a hard disk drive
• Disk drives have peculiar characteristics
  – Long, and worse, variable access latencies
  – Accesses performed in chunks of fixed size
    • Atomicity only for accesses of that size
  – Highly variable performance depending on exactly what gets put where
  – Unpleasant failure modes
• So the operating system needs to smooth out these oddities
What Does That Lead To?

• Great effort by file system component of OS to put things in the right place on a disk
• Reordering of disk operations to improve performance
  – Which complicates providing atomicity
• Optimizations based on caching and read-ahead
  – Which complicates maintaining consistency
• Sophisticated organizations to handle failures
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
For Example,

- A CPU
- It has a program counter register indicating where the next instruction can be found
  - An instruction reference
- It supports a set of instructions
  - Its repertoire
- It has contents in registers and RAM
  - Its environment
Another 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 almost always are multiple processes
• The OS has a certain amount of physical memory
  – To hold the environment information
• There is usually only one set of registers
• The process doesn’t have exclusive access to the CPU
  – Due to 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 Links

• A communication link allows one interpreter to talk to another
  – On the same or different machines
• At the physical level, wires and cables
• At more abstract levels, networks and interprocess communication mechanisms
• Some similarities to memory abstractions
  – But also differences
Basic Communication Link Operations

- **SEND**(link\_name, outgoing\_message\_buffer)
  - Send some information contained in the buffer on the named link

- **RECEIVE**(link\_name, incoming\_message\_buffer)
  - Read some information off the named link and put it into the buffer

- Like WRITE and READ, in some respects
Why Are Communication Links Distinct From Memory?

• Highly variable performance
• Potentially hostile environment for the operations
• Generally asynchronous
• Receiver may only perform the operation because the SEND occurred
  – Unlike a typical READ
• No necessary guarantee of delivery
An Example Communications Link

• A Unix-style socket
• SEND interface:
  – send(int sockfd, const void *buf, size_t len, int flags)
  – The sockfd is the link name
  – The buf is the outgoing message buffer
• RECEIVE interface:
  – recv(int sockfd, void *buf, size_t len, int flags)
  – Same parameters as for send
What About Those Other Socket Parameters?

• The `len` and `flag` fields?

• A common attribute of instances of abstractions
  – Especially higher level versions

• They provide additional semantics specific to the abstraction

• Generally improving the power of the higher level abstraction
Implementing the Communications Link Abstraction in the OS

• A bit trickier than the memory and interpreter abstraction, in some cases

• Unlike those, the OS does not have full control of what’s going on

• The network doesn’t belong to the OS
  – Only its own network interface does

• Another entity is often doing half the work
  – Typically another machine’s OS
What Are the Implications?

• Greater uncertainty about the outcome of an operation
  – Things fail for reasons our OS can’t see or learn

• Greater asynchrony
  – The remote OS might not regard the operations as equally important as our OS does

• Higher possibilities for security problems
  – Remote OS not equally trusted
  – Network between the two potentially untrustworthy
What Do We Do About Those Issues?

• OS must be prepared for likely failures
• And high degrees of asynchrony
  – Bad idea to block entire system while waiting for the network
• OS shouldn’t have complete trust in what comes in from the network
  – But often the OS is in no position to determine its trustworthiness
Some Other Abstractions

• Actors
  – Users or other “active” entities
• Virtual machines
  – Collections of other abstractions
• Protection environments
  – Security related, usually
• Names
• Not a complete list
• Not everyone would agree on what’s distinct
System Services for OSes

- One major role of an operating system is providing services
  - To human users
  - To applications
- What services should an OS provide?
An Object Oriented View of OS System Services

• Services are delivered through objects
  – Can be instantiated, named, and destroyed
  – They have specified properties
  – They support specified methods

• To understand a service, study its objects
  – How they are instantiated and managed
  – How client refers to them (names/handles)
  – What a client can do with them (methods)
  – How objects behave (interface specifications)
Typical OS System Service Types

- Execution objects
  - Processes, threads, timers, signals
- Data objects
  - Files, devices, segments, file systems
- Communications objects
  - Sockets, messages, remote procedure calls
- Protection objects
  - Users, user groups, process groups
- Naming objects
  - Directories, DNS domains, registries
System Services and Abstractions

- Services are commonly implemented by providing appropriate abstractions
- For example,
  - The service of allowing user code to run in a computing environment
  - Requires a couple of abstractions, at least:
    - The virtual environment abstraction
    - The process abstraction
The Virtual Environment Abstraction

• A CPU executes one program at a time
  – It is a serially reusable resource
• But we want to run multiple programs “simultaneously”
  – Without them treading on each other’s toes
• A good way to do that is to build a virtual execution environment abstraction
  – Make it look like each program has its own computer
What Should This Abstraction Provide?

• Each program should see its own resource set
  – A complete virtual computer with all elements
    • CPU
    • Memory
    • Persistent storage
    • Peripherals

• Isolation from other activities
  – Including non-related OS activities

• Each program should think it has the real machine to itself
How To Do That?

• We won’t go into detail now
  – But will later
• In essence, the OS must multiplex its real resources
  – Among the various process’ virtual computers
• Requiring care in saving and restoring state
• And attention to fair use and processes’ various performance requirements
The Process Service

• Given we want per program virtual environments,
• We need an interpreter abstraction that provides the ability to run user code
  – The process
• With some very useful properties:
  – Isolation from other code
  – Isolation from many system failures
  – Guarantees of access to certain resources
• Processes can communicate and coordinate
  – But do so through the OS
  – Which provides isolation and synchronization
What Is a Process?

• An interpreter that executes a single program
  – It provides illusion of continuous execution
  – Despite fact that the actual CPU is time-shared
    • Runs process A, then process B, then process A

• What virtual environment does a program see?
  – Programs don't run on a real bare computer
  – They run inside of a process
  – Process state is saved when it is not running
  – Process state is restored when it runs again
Processes and Programs

• Program = set of executable instructions
  – Many processes can run the same program

• Process = executing instance of program
  – It has saved state
    • Memory, contents, program counter, registers, ...  
  – It has resources and privileges
    • Open files, user-ID, capabilities, ...  
  – It may be the unit of CPU sharing
    • CPU runs one process, then another
Problems With the Process Abstraction

• Processes are very expensive
  – To create: they own resources
  – To dispatch: they have address spaces

• Different processes are very distinct
  – They cannot share the same address space
  – They cannot (usually) share resources

• Not all programs want strong separation
  – Cooperating parallel threads of execution
  – All are trusted because they run same code
So the Process Abstraction Isn’t Sufficient

• To meet common user needs
• What if I have a program that can do multiple things simultaneously?
• And requires regular, cheap communications between those different things?
• Processes are too expensive
• And make regular communications costly
• So I need another abstraction
Threads

• An abstraction built on top of the process abstraction
• Each process contains one or more threads
• Each thread has some separate context of its own
  – Like a program counter and scheduling info
• But otherwise shares the resources of its process
• Threads within a process can thus communicate easily and cheaply
Characteristics of Threads

• Strictly a unit of execution/scheduling
  – Each thread has its own stack, PC, registers

• Multiple threads can run in a process
  – They all share the same code and data space
  – They all have access to the same resources
  – This makes the cheaper to create and run

• Sharing the CPU between multiple threads
  – User level threads (with voluntary yielding)
  – Kernel threads (with preemption)
Using the Abstractions

- When a programmer wants to run code, then, he can choose between abstractions.
- Does he want just a process?
- Or does he want a process containing multiple threads?
- Or perhaps multiple processes?
  - With one thread each?
  - With multiple threads?
When To Use Processes

- When running multiple distinct programs
- When creation/destruction are rare events
- When running programs (even instances of the same code) with distinct privileges
- When there are limited interactions and few shared resources
- When you need to prevent interference between programs
  — Or need to protect one from failures of the other
An Example of Choosing Processes

• When implementing compilation in a shell script

```cpp
$1.c | cc1 | ccopt > $1.s
as $1.s
ld /lib/crt0.o $1.o /lib/libc.so
mv a.out $1
rm $1.s $1.o
```

• Each of these programs gets a separate process
Why?

- The activities are serial
- The only resources to be shared are through the file system
- Failure of one program could damage the others if too much is shared
  - Who knows what `rm` might get rid of, for example?
When To Use Threads

- When there are parallel activities in a single program
- When there will be frequent creation and destruction
- When all activities can run with same privileges
- When they need to share resources
- When they exchange many messages/signals
- When there’s no need to protect them from each other
An Example for Choosing Threads

• A web server
• Multiple users will request service
• Desirable to share much of the server data
  – Such as copies of pages many users want to see
  – And information about overall load and performance
• But the pages can be served to users in parallel
  – In particular, if serving one user’s page is slow, don’t slow down other users
Which Abstraction To Choose?

• If you use multiple processes
  – Your application may run much more slowly
  – It may be difficult to share some resources

• If you use multiple threads
  – You will have to create and manage them
  – You will have serialize resource use
  – Your program will be more complex to write
  – You may get weird bugs

• TANSTAAFL
  – There Ain't No Such Thing As A Free Lunch
Generalizing the Concepts

- There are many other abstractions offered by the OS
- Often they provide different ways of achieving similar goals
  - Some higher level, some lower level
- The OS must do work to provide each abstraction
  - The higher level, the more work
- Programmers and users have to choose the right abstractions to work with
Abstractions and Layering

• It’s common to create increasingly complex services by layering abstractions
  – E.g., a file system layers on top of an abstract disk, which layers on top of a real disk

• Layering allows good modularity
  – Easy to build multiple services on a lower layer
    • E.g., multiple file systems on one disk
  – Easy to use multiple underlying services to support a higher layer
    – E.g., file system can have either a single disk or a RAID below it
A Downside of Layering

• Layers typically add performance penalties
• Often expensive to go from one layer to the next
  – Since it frequently requires changing data structures or representations
  – At least involves extra instructions
• Another downside is that lower layer may limit what the upper layer can do
  – E.g., an abstract disk prevents disk operation reorderings to maximize performance
Layer Bypassing

• Often necessary to allow a high layer to access much lower layers
  – Not going through one or more intermediaries
• Most commonly for performance reasons
• If the higher layer plans to use the very low level layer’s services,
  – Why pay the cost of the intermediate layer?
• Has its downsides, too
  – Intermediate layer can’t help or understand