Important OS Properties

- For real operating systems built and used by real people
- Differs depending on who you are talking about
  - Users
  - Service providers
  - Application developers
  - OS developers

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

Upward Compatibility

- People want new releases of an OS
  - New features, bug fixes, enhancements
  - Security patches to protect from malware
- 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

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

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/…
    - May even run on Windows platforms
- An ABI compliant program will run (unmodified) on any compliant system

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
- Basic requirements will change many times
- Support costs will dwarf initial development
- This makes maintainability critical
- Aspects of maintainability:
  - Understandability
  - Modularity/modifiability
  - Testability

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

An Example Memory Abstraction

- 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

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 overriden

An Example Interpreter Abstraction

- 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

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

An Example Communications Link Abstraction

- 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

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