Operating System Structures

- While process management, memory management, file systems, and I/O provide an idea of what an operating system *does* (its verbs), additional concepts help define what an operating system is *made of* (its nouns)
- Three different perspectives for these concepts:

users	user interface, programs ("system programs," "system utilities," "application programs")
programmers	application programming interfaces (APIs), system calls
operating system designers	mechanisms, policies, layers, microkernels, modules, virtual machines

User Interfaces to Operating Systems

- Command interpreter or *shell*
  - Text-driven, command-response interface style
  - A shell is ultimately just a program, so there may be more than one
  - Two variations: embed system calls in shell, or separate all system calls as external programs (keeps the shell small, and protects it from operating system changes)
- Graphical user interface
  - Menu-driven and/or direct manipulation interface style
  - Also "just a program," so there may also be more than one GUI environment available
- In many cases, both UI types are provided; really an orthogonal issue to the operating system itself

Programs: "System," "Utility," "Application"

- Primarily an end-user distinction; they're all the same to the operating system
- "Application programs" generally refer to the programs that directly perform the work that we need to do: e-mail, Web browsing, document creation, etc.
- When a program's work involves something on the computer itself, it may be viewed as a "system utility"
- Finally, programs whose functions correspond most directly with an operating system's underlying services may be viewed as "system programs"

APIs and System Calls

- "Beneath" the programs that end-users run are *application programming interfaces* (APIs) — functions that the programs' developers invoke
- APIs themselves are implemented by another layer of developers; ultimately, they invoke an OS's *system calls*
    - ◇ APIs hide system call-specific details from your average programmer; they keep semantics at the level of the programming language and may facilitate portability (e.g., standard C interfaces), but not always (e.g., Windows APIs)
- System calls define the direct programming interface to operating system services, and form the boundary between user and kernel modes

Operating System Design and Implementation

- In the end, operating systems are software programs, and are as subject to good software engineering practices and principles as any other program
- Interesting side reading: *The Art of Unix Programming* by Eric S. Raymond (http://www.faqs.org/docs/artu)
- Also notable: *The Mythical Man-Month* by Frederick P. Brooks (primarily about the software development process, but the software in question is an operating system, IBM's OS/360)

Mechanisms and Policies

- A *mechanism* defines *how* something is done; a *policy* states *what* is actually done
- General principle: separate mechanism from policy; or, allow for a single mechanism to support the widest possible range of policies (i.e., a change in policy should not necessitate a change in mechanism)
- Good principle to follow in all software design, but particularly important in operating systems
- Yet another way: separate the *interface* (policy) from the *engine* (mechanism)

Operating System Implementation

- Originally machine or assembly language
- Feasible these days in a higher-level language, such as C or C++ — allows for (potentially) better portability and improvements based on compiler technology
- C and C++ have reigned for quite a while; some research has involved going beyond these languages (Java for "native" object-orientation, Haskell for the benefits of functional languages)
- Subject to possible inefficiencies; frequently coupled with changes at the hardware level

Interesting Operating System Language Choices

Or, "neither assembly nor C/C++" :)

Operating System	Implementation	Era
Master Control Program (MCP)	ESPOL (ALGOL variant)	1960s
Multics	PL/1	1960s–1980s
Hello	Standard ML	1999 (master's thesis)
House, Osker	Haskell	present (research)

Overall Operating System Structure

- The usual rules apply: we want easy modification, robust operation, and efficiency (speed)
- Simplest case first: *monolithic structure* (MS-DOS, early Unix versions)
    - Hardware below, programs above, no other distinctions in between
    - MS-DOS (and other early PC operating systems) had it even worse — hardware didn't support dual-mode operation, so many protections taken for granted today weren't even available

Layered Approach

- Strict separation of functions and data structures; "layer zero" represents the hardware
- Each layer may only call functions and use data structures from itself and the layers below it
- Benefits: the usual "good things" that come from abstraction, information hiding, and isolation
- Drawbacks: strict top-down approach makes the actual layers hard to define — cyclic dependencies between functions must be avoided or else layer separation can't be done; possible efficiency issues as well

Microkernels

- Minimalist approach to the kernel — include only what is absolutely necessary, and everything else is a program in user space
- Introduced by CMU in the *Mach* operating system
- Services communicate using *message passing*
- Benefits: ease of OS extension and porting; somewhat enhanced security because more code is in user space
- Drawbacks: performance issues due to increased overhead (message passing, fine-grained separation)

Modules

- Current "best-of-both-worlds" approach, particularly for existing Unix derivatives such as Solaris, Linux, and Mac OS X (Darwin)
- What matters is the abstraction: module-based kernels publish well-defined interfaces to their services
- Developers expand kernel functionality by adding modules that "plug into" the relevant interfaces
  - Solaris: 7 types of loadable modules
  - Mac OS X (Darwin): Mach microkernel is actually one of the components inside the kernel

Virtual Machines

- The ultimate abstraction: a user-mode program that runs an operating system kernel
- Device drivers in the virtual machine actually connect to hardware abstractions provided by the virtual machine software; for example, a "disk drive" in the virtual machine may map to a file in the physical host
- Dual-mode simulation: virtual user and kernel modes that are both in user mode on the physical host
- Virtual environment may go as far as translating machine instructions, but not necessarily

Notable Virtual Machine Implementations

- *VMware*: x86 virtual machine supporting multiple operating systems
- *User Mode Linux*: Runs Linux kernel as a user process
- *VirtualPC*: x86 virtual machine on Mac OS X, all the way down to the CPU; translates PowerPC instructions to x86, but provides some PowerPC-native implementations of some functions
- *Java*: special bytecode format to represent code, with a just-in-time (JIT) compiler translating into native code

Operating System Generation and System Boot

- In the end, operating systems are ultimately sets of files, built from source code
- Installation sometimes requires customization, based on the installation target's hardware and devices
- A restart (warm or cold) points the CPU to start executing from a predetermined location
  - For large, general purpose OSes, this initial program is a *bootstrap program* or *loader* (e.g., BIOS, OpenFirmware, EFI [Extensible Firmware Interface]) that locates the rest of the OS in secondary storage and loads/runs it, usually from a known *boot block*
  - In other systems, the predetermined start location *is* the start of the operating system's code (firmware); other variations include booting off the network

Exercise: Build an Operating System Kernel

- It's easier than you think!
- Easily obtainable kernel sources:
  - Linux (of course)
  - XNU (a Mach/BSD kernel; a.k.a. the Darwin or Mac OS X kernel)
- In addition to the sources, you will need: developer tools (compile/make) and instructions
- Finally, note how it's just the beginning — many more files are involved before you have a "full" OS