HARDWARE DETAILS
OS DESIGN PATTERNS

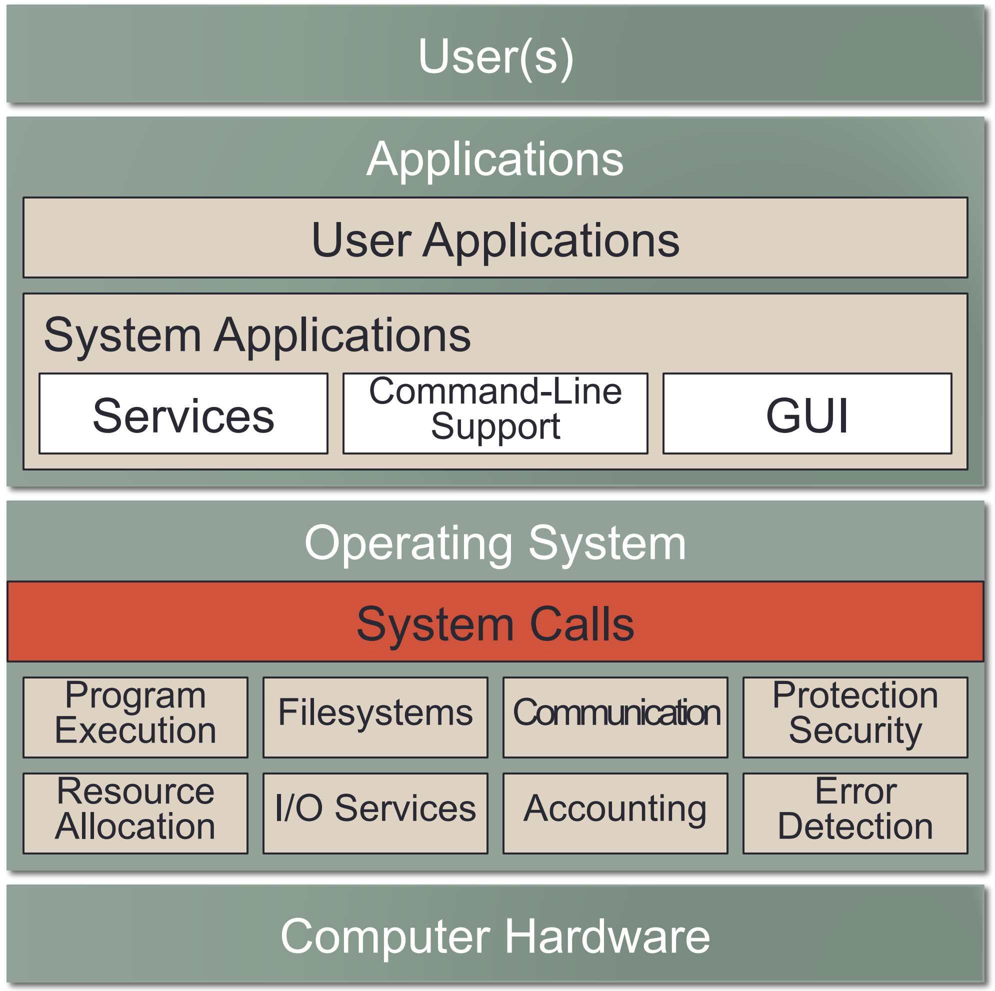
CS124 – Operating Systems

Spring 2024, Lecture 3

Operating System Components

- Operating systems commonly provide these components:
- Last time:
  - Applications can’t access operating system state or code directly
- OS code and state are stored in kernel space
  - Must be in kernel mode to access this data
- Application code and state is in user space

- How does the application interact with the OS?



Operating Modes and Traps

- Typical solution: application issues a **trap** instruction
  - A trap is an intentional software-generated exception or interrupt
  - (as opposed to a fault; e.g. a divide-by-zero or general protection fault)
- During OS initialization, the kernel sets up a handler to be invoked by the processor when the trap occurs
  - When the trap occurs, the processor changes the current protection level (e.g. switch from user mode to kernel mode)
- Benefits:
  - Applications can invoke OS routines without having to know what address they live at
    - (and the location of OS routines can change when the OS is upgraded)
  - OS can verify applications' requests and prevent misbehavior
    - e.g. check arguments to system calls, verify process permissions, etc.

IA32: Operating Modes and Traps

- On IA32, programs use the **`int`** instruction to cause a software interrupt (a.k.a. software trap)
  - e.g. **`int $0x80`** is used to make Linux system calls
- On IA32, different operating modes have different stacks
  - Ensures the kernel won't be affected by misbehaving programs
- Arguments to the interrupt handler are passed in registers
  - Results also come back in registers
- Example:

```
movl $20, %eax          # Get PID of current process
int $0x80               # Invoke system call!
# Now %eax holds the PID of the current process
```

- Of course, operating systems provide wrappers for this

```
int pid = getpid();  /* Does the syscall for us */
```

IA32: Interrupt Descriptor Table

- IA32 uses an Interrupt Descriptor Table (IDT) to specify handlers for interrupts
  - IA32 supports 256 interrupts, so the IDT contains 256 entries
- In protected mode, each interrupt descriptor is 8 bytes
  - Specifies the address of the handler (6 bytes), plus flags (2 bytes)
  - (Privilege level that the handler runs at is specified elsewhere in the IA32 architecture; will revisit this topic in the future)
- When a program issues an **`int`** ***`n`*** instruction:
  - Processor retrieves entry *n* in the Interrupt Descriptor Table
  - If the caller has at least the required privilege level (specified in the IDT entry), the processor transfers control to the handler
    - (if privilege level changes, the stack will be switched as well)
  - The handler runs at the privilege level it requires (e.g. kernel mode)

IA32: Interrupt Descriptor Table (2)

- When interrupt handler is finished, it issues an `iret`
  - Performs the same sequence of steps in reverse, ending up back at the caller's location and privilege level
    - (if the privilege level changes, the stack will be switched again)
- Location of Interrupt Descriptor Table can be set with the `lidt` IA32 instruction
  - Can only be executed in kernel mode (level 0)
  - Operating system can configure the IDT to point to its entry-point(s)
- If you're wondering:
  - IA32 processors start out in kernel mode when reset
  - Allows OS to perform initial processor configuration (interrupt handlers, virtual memory, etc.) before switching to user mode
  - Will discuss bootstrap sequence in much more detail next week

Exceptional Control Flow

- **Exceptional control flow** is very important for many other OS facilities
- Already saw **traps**
  - Intentional, software-generated exceptions
  - Frequently used to invoke operating system services
  - Returns to next instruction
- **Interrupts** are caused by hardware devices connected to the processor
  - Signals that a device has completed a task, etc.
  - The current operation is interrupted, and control switches to the OS
  - OS handles interrupt, then go back to what was being done before
  - Returns to next instruction

Exceptional Flow Control (2)

- **Faults** are (usually unintentional) exceptions generated by attempting to execute a specific instruction
  - Signals a potentially recoverable error
    - e.g. a page fault generated by the MMU
    - e.g. a divide-by-zero fault generated by the ALU
  - Returns to the current instruction, if the OS can recover from fault!
- **Aborts** are nonrecoverable hardware errors
  - Often used to indicate severe hardware errors
    - e.g. memory parity errors, system bus errors, cache errors, etc.
  - Doesn't return to interrupted operation

IA32 Aborts

- IA32 machine-check exception is an abort
  - Signaled when hardware detects a fatal error
- IA32 also has a **double-fault** abort
- Scenario:
  - User program is running merrily along… then an interrupt occurs!
  - CPU looks in Interrupt Descriptor Table (configured by OS) to dispatch to interrupt handler
  - When CPU attempts to invoke the handler, a fault occurs!
    - e.g. a general protection fault, because the handler address was wrong
  - Double-fault indicates that a fault occurred during another fault
- Another scenario that causes a double-fault:
  - User program causes a page fault…
  - When CPU tries to invoke page-fault handler, it causes *another* page-fault to occur. Rats.

IA32 Aborts (2)

- Double-faults are signs of operating system bugs
  - It should be impossible for a user-mode program to cause one
- Of course, IA32 allows operating systems to register a double-fault handler in the IDT (interrupt 8)
- If a fault occurs while attempting to invoke the double-fault handler, a **triple-fault** occurs
- At this point, the CPU gives up.
  - Enters a shutdown state that can only be cleared by a hardware reset, or (in some cases) by certain kinds of interrupts
- You may see double- or triple-faults during the term ☺
  - Triple-faults in Pintos can manifest by the machine rebooting over and over again in the emulator

Multitasking and Hardware Timer

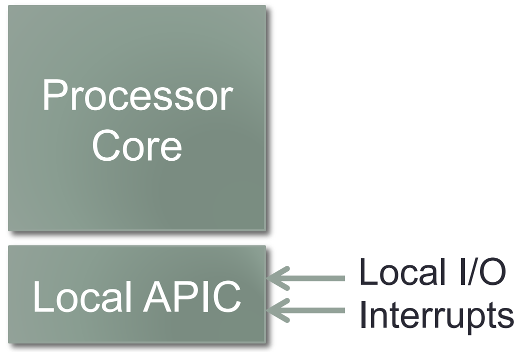
- Certain kinds of multitasking rely on hardware interrupts
- Early multitasking operating systems provided **cooperative multitasking**
  - Each process voluntarily relinquishes the CPU, e.g. when it blocks for an IO operation to complete, when it yields, or terminates
  - (This would implicitly occur when certain system calls were made)
- Problem:
  - Operating system is susceptible to badly designed programs that never relinquish the CPU (e.g. buggy or malicious program)
- A number of OSes only provided cooperative multitasking:
  - Windows before Win95
  - MacOS before MacOS X
  - <u>Many</u> older operating systems (but generally not UNIX variants!)

Multitasking and Hardware Timer (2)

- Solution: incorporate a **hardware timer** into the computer
- OS configures the timer to trigger after a specific amount of time
  - When time runs out, the hardware timer fires an interrupt
  - OS handles the interrupt by performing a context-switch to another process, then starting the timer again
- Operating systems that can interrupt the currently running process provide **preemptive multitasking**
- Obviously, all hardware timer configuration must require kernel-mode access
  - Otherwise, a process could easily take all the time it wants
- Virtually all widely used OSes now have preemption
- (All UNIX variants have basically always had preemption)

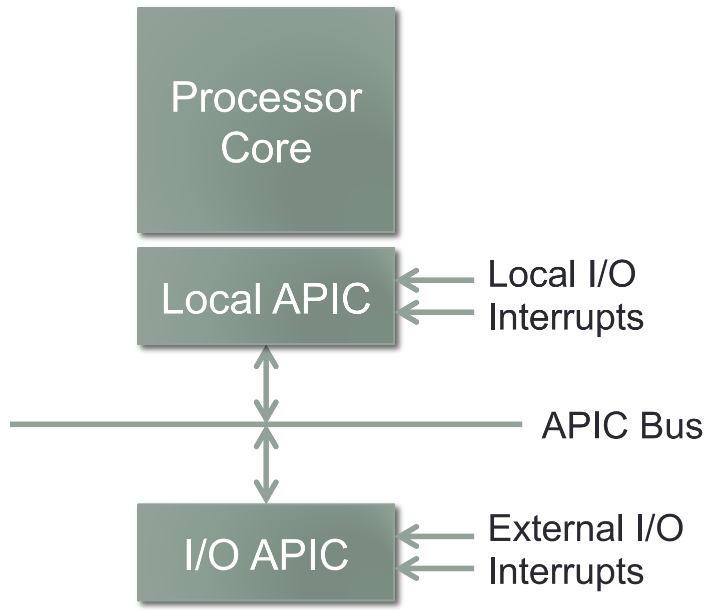
IA32 APIC

- IA32 has a component for handling hardware interrupts: the Advanced Programmable Interrupt Controller (APIC)
  - Software interrupts are handled within the processor core itself
- The APIC handles many different source of interrupts
  - The APIC is itself broken down into several different components
- The Local APIC handles:
  - Interrupts from locally connected I/O devices (single-core processors only)
  - Timer-generated interrupts
  - Thermal sensor interrupts
  - Performance-monitoring interrupts
  - Interrupts caused by APIC errors
- All of these interrupt sources have various config options



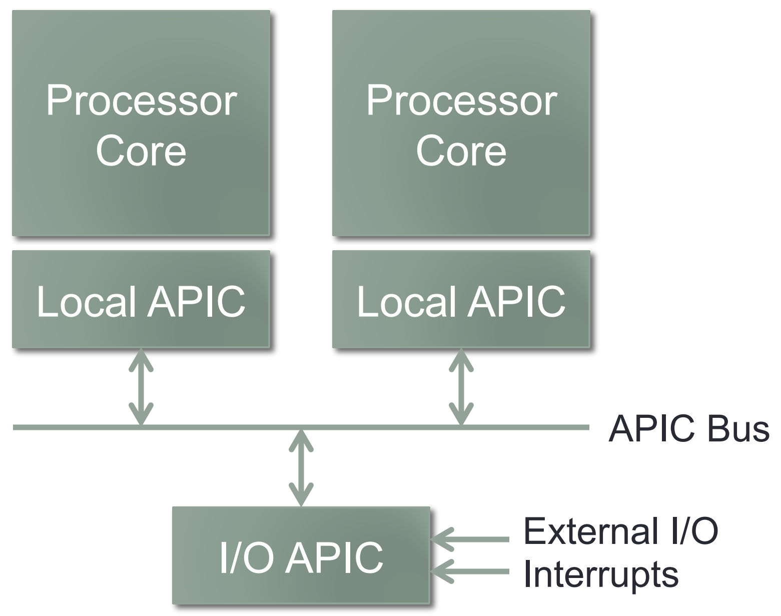
IA32 APIC (2)

- The I/O APIC handles interrupts from additional external devices
  - Allows specific devices to be mapped to specific interrupt numbers
  - Allows interrupts from different devices to be prioritized
    - (e.g. if two devices signal an interrupt at the same time, which interrupt should be handled first?)
- The I/O APIC is typically part of the motherboard chipset, not the CPU



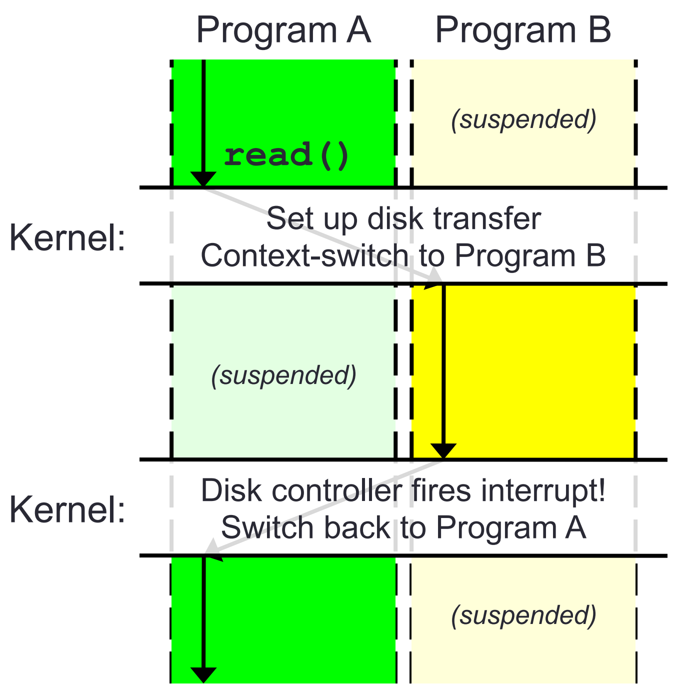
IA32 APIC (3)

- On a multiprocessor/multicore system:
  - Local APIC no longer has local I/O interrupts
  - All I/O interrupts come from the I/O APIC via the APIC bus
- I/O APIC can route external interrupts to specific processors, based on OS configuration
- Processors can also send **inter-processor interrupts** (IPIs) to each other
  - Every Local APIC has its own ID
  - Can issue an IPI to a specific CPU
  - Can issue an IPI to self
  - Can issue an IPI to everyone
  - Can issue an IPI to “everyone but self”



Long-Running I/O Operations

- Hardware interrupt handling is also very important with long-running I/O operations (and most of them are)
- Example: hard disk I/O
  - e.g. reading a block from a magnetic disk can take 3-15ms
  - e.g. reading a block from an SSD can take several μs
- When a process performs a long-running I/O operation:
  - Process is unable to proceed until the I/O operation completes…
  - OS sets up the I/O operation, then switches to another process that can run
  - When I/O operation is finished, the hardware fires an interrupt to notify the OS



Summary: Hardware Requirements

- Modern operating systems rely heavily on these hardware facilities to provide their functionality:
  - At least dual-mode execution: kernel mode and user mode
  - Virtual memory management, including memory protection to enforce barrier between kernel space and user space
  - Ability to trap (or otherwise transfer control) into kernel-mode code
  - Support for both software interrupts (e.g. traps, faults) and hardware interrupts (e.g. I/O devices)
  - A hardware timer facility to allow OS to regain control of the CPU
- We will discuss all of these hardware capabilities in more detail in the future

OS Structural Patterns

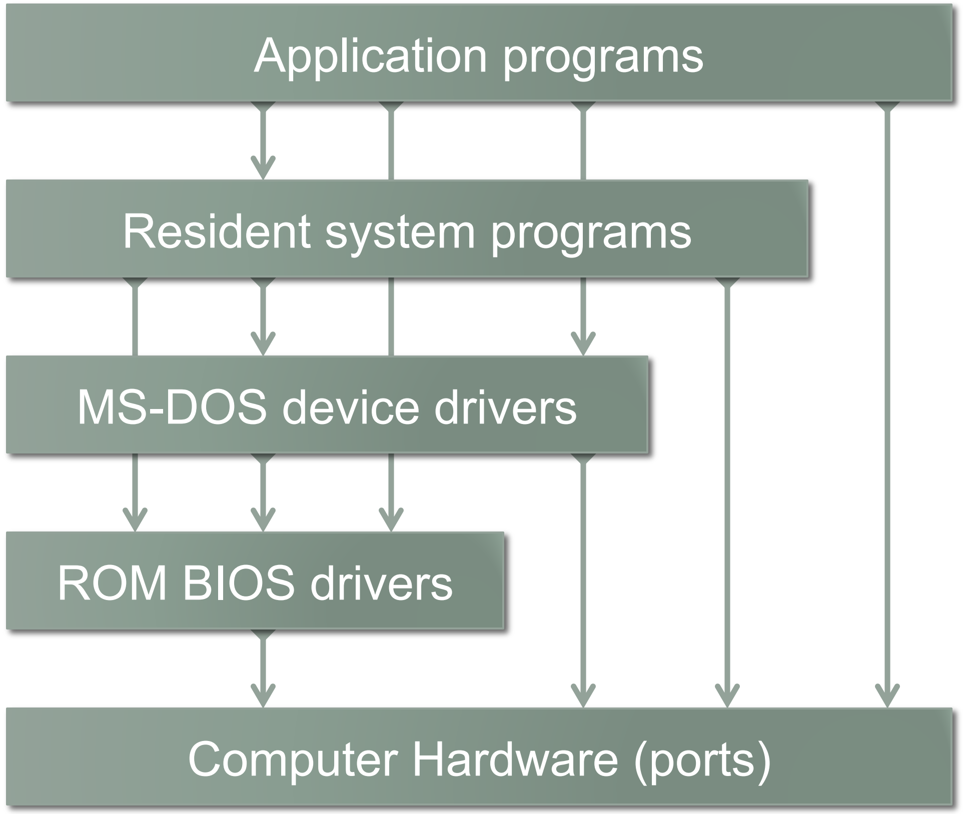
- A guiding principle in OS design: **separation of policy and mechanism**
- **Policy** specifies what needs to be done
  - e.g. Which virtual memory page should the OS evict?
  - e.g. What process should the OS run next?
- **Mechanism** specifies how to do it
  - e.g. how to save dirty pages, how to update the CPU's page table to reflect the page-out/page-in operations, other bookkeeping
  - e.g. how to save the current process' context, how to restore the new process' context
- Mechanisms are unlikely to change substantially over time
  - (In the context of a given OS and set of hardware)
- Policies are <u>very</u> likely to change
  - May even be part of an operating system's configuration options!

OS Structural Patterns (2)

- Operating systems can follow various structural patterns
- **Simple structure:**
  - OS code is not cleanly divided into modules
  - Every part of the OS can access every other part of the OS
  - (Often, everything can also access the computer hardware)
- Usually happens when OS starts out as a simple, limited system and then grows well beyond its original scope
- Problems:
  - Highly susceptible to both operating system and application bugs
  - Often very easy for malicious programs to compromise the OS
  - Tends to be very difficult to extend the OS' functionality

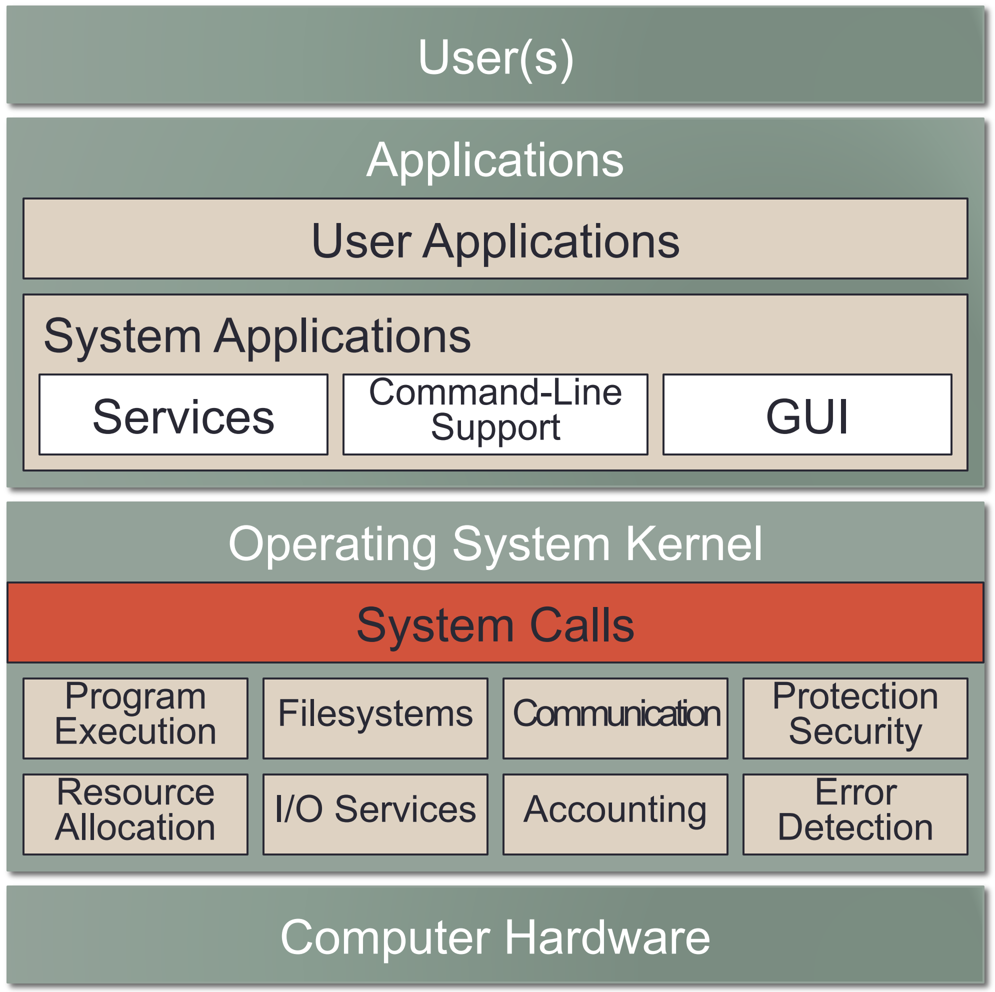
Simple Structure: MS-DOS

- Written for 8086/8088 processors, which had no protected mode execution
- Everything could access everything else in the system (and often did)
- Applications and resident programs could control hardware
  - e.g. computer games with specialized graphics modes
  - e.g. extended-memory libraries
- Malicious programs could directly manipulate the OS
  - e.g. stealth viruses would intercept DOS `int 21` trap to make themselves invisible



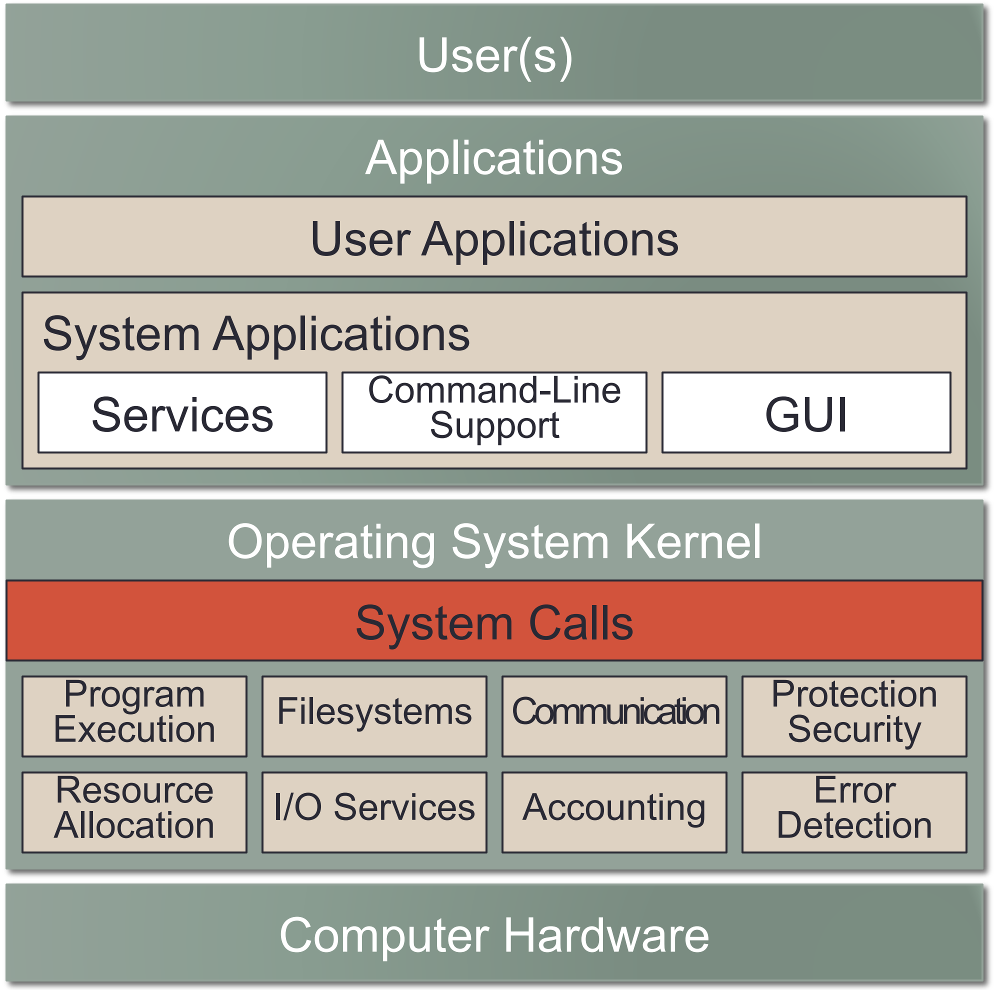
OS Structural Patterns: UNIX

- Initial versions of UNIX were slightly more structured than MS-DOS
- OS is divided into kernel and system applications
- Initially, the kernel itself was largely unstructured
- Also, a large amount of OS functionality was built into the kernel itself
  - Called a **monolithic kernel**
  - Amount of code that runs in kernel mode is very large
  - More chance for bugs to have a severe impact



OS Structural Patterns: UNIX (2)

- As with MS-DOS, kernel subsystems could access any other subsystem
  - Kernel runs in protected mode, so applications can't compromise the system easily…
  - However, still susceptible to severe bugs, OS crashes
  - Hard to maintain and extend
- Big benefit: it's fast!
  - Easy for kernel operations to directly access whatever state or functions they need
  - Many modern OSes include monolithic aspects in their implementations



Layered Structure

- Want to avoid the issues of monolithic operating systems
  - Introduce more structure into the design, in one way or another
- An early approach: **layered structure**
- Each layer encapsulates its own state, and exposes a set of operations to higher layers
- Higher layers rely solely on the operations exposed by lower layers
  - They don't care about implementation details of lower layers, only the abstractions that the lower layers expose
- Layers can be tested in isolation of each other, starting with the lowest layers
  - Once lower layers are validated, testing can begin on higher layers

Layered Structure: The THE OS

- First example: the THE multiprogramming system
  - Designed by a team lead by Edsger Dijkstra, at the Technische Hogeschool Eindhoven (THE) in 1965-1968
  - A batch processing system that supported multitasking
- Divided the operating system into 6 layers
- Layer *i* basically interacts only with layer *i* – 1
  - Mostly true for lower layers, but higher layers might interact with multiple lower layers
- Layers implemented in bottom-up order; start with layer 0
- Each layer was heavily tested. Consequences:
  - To test layer *i*, at least some of layer *i* + 1 had to be implemented
  - To test each layer exhaustively, implementers spent a lot of time planning and reviewing their tests to ensure they were complete
  - (they were using a batch-processing mainframe after all)

Layered Structure: The THE OS (2)

- Each layer provided an abstraction for the higher layers to rely upon, simplifying the implementation
- Layer 0: multiprogramming facilities
  - CPU scheduler, including support for interrupt handling, tasks blocked on semaphores, context switches
- Layer 1: virtual memory system, pager
- Layer 2: communication between OS and console
- Layer 3: device I/O, including buffering
- Layer 4: user programs
  - The THE OS only had 5 processes, for compiling, running, and printing output from user programs
- Layer 5: the operator (Dijkstra: “not implemented by us”)

Layered Structure: The THE OS (3)

- Resulted in a very reliable OS with a low rate of bugs
  - Each layer was basically reliable when next layer was being tested
  - Bugs tended to be very easy to identify and fix, just by inspection
  - This was good – they didn't have debugging facilities for the machine they were programming
- Designing/implementing an OS this way takes a long time
  - A lot of time spent up-front designing interfaces between the layers
  - Makes it <u>much</u> easier to identify and isolate bugs, though
  - Might actually save companies time in the long run, but companies are usually afraid of taking that risk
- Layered approach can greatly reduce OS performance
  - Interactions with the OS often require calls across many layers
- Tradeoff: make fewer layers with greater functionality
  - Still realizes many of the benefits of layering

Modular Kernels

- Besides reliability issues, monolithic kernels also have extensibility issues
  - e.g. want to support a wide range of devices, filesystems, etc.
- Example: add support for a new device to monolithic OS
  - Must add the device driver to the kernel's code-base, then recompile the entire kernel
- Leads to several problems:
  - Kernel becomes huge, because it must include compiled-in support for all supported devices, filesystems, etc.
  - More code = more bugs, which makes kernel even less reliable
- Modern monolithic kernels use **loadable kernel modules** to provide extensibility during normal operation
  - Such kernels are called **modular kernels**

Modular Kernels (2)

- Modular kernels always include certain core services:
  - e.g. CPU scheduling, virtual memory management, inter-process communication are always compiled into the kernel
- Kernel defines interfaces for parts of kernel that require extensibility
  - e.g. what set of operations must all filesystems support?
- Kernel includes a **module loader** that is able to load and statically link a kernel module directly into the kernel
  - At load time, module-references to kernel symbols are updated with actual addresses of kernel variables and functions
  - (Modules can also expose symbols for other modules to access)
  - Module code is loaded into kernel space; code runs in kernel mode
  - Modules can be unloaded, if no other module is using its symbols

Modular Kernels (3)

- Benefits:
  - Largely retains high performance of monolithic kernels
  - Modules can control what symbols they expose, and encapsulate critical module state
  - Yields a much smaller monolithic kernel, since only the modules required for proper functionality are loaded into memory
- Drawbacks:
  - Only slightly reduces reliability issues!
  - Modules run in kernel mode; bugs can still cause the OS to crash
- Generally, modules reference and expose symbols that are <u>global</u> within the kernel
  - A modular kernel doesn’t necessarily have a more well-defined structure; it’s just easier to extend the kernel’s facilities

Modular Kernel Notes

- Modular kernels still frequently compile certain modules into the core kernel
  - e.g. ATA drivers, SCSI drivers, filesystems commonly used with OS
  - Often a requirement to make the boot process more straightforward
- The term “monolithic kernel” refers primarily to how much OS code runs in kernel mode vs. user mode
  - By now, virtually all monolithic kernels are also modular

Next Time

- Continue discussion of OS design patterns