Operating Systems Overview

No single definition, but many perspectives:

- *Role in an overall system:* Intermediary between computer hardware and “everything else”
- *User view:* Provides an environment, preferably *easy to use*, *convenient*, and *efficient*, in which users can work
- *System view:* Acts as *resource allocator* for the finite components of a system; also a *control program* that runs all other programs and manages devices
- *Textbook definition:* “The one program running at all times on the computer” (a.k.a. the *kernel*)

Quick Review: Computer System Organization

- Four primary components: *central processing unit(s) (CPUs)*, *device controllers*, *common bus*, *shared memory*
- System startup points the CPU to a *bootstrap program* typically stored in *firmware* that performs low-level initialization of components
- The bootstrap program locates the operating system kernel, loads it into memory, and jumps to its first instruction; execution proceeds from there
- During operation, events are signaled by *interrupts* which divert the processor to a *service routine*

Storage Structure

- Classic *von Neumann* architecture:
  1. Fetch next instruction from memory
  2. Decode and execute instruction
  3. Primary activities: load, modify, store
  4. Rinse and repeat
- Recent theory students take note — ultimately not that different from a Turing machine
- In the ideal model, we have a single, permanent, infinite *main memory* store; in practice, this is not possible

- Real-world systems *stratify* memory into a *storage hierarchy*, driven by one rule of thumb: the faster the memory, the more expensive it is
- The fastest but costliest — and therefore smallest — memory resides closest to the CPU
- The "farther" from the CPU, the slower but less expensive — and therefore larger
- Typical sequence: *registers* » *cache* » *main memory* » *disks of different types* » *other storage (tapes, etc.)*
- Additional factor: typically, memory in the upper levels is also *volatile* — it loses content when it loses power
- Net result: the storage hierarchy requires additional management, from moving memory among levels to ensuring that volatile data reaches nonvolatile storage

Input/Output (I/O) Structure

- Computer systems are not closed worlds; they require *devices* that connect from (*input*) and/or to (*output*) the real (physical) world
  - Storage technologies beneath main memory (disks, flash drives, CD/DVD drives) typically count among these external devices
- Each device has a *device controller*; the controller communicates with the CPU and/or main memory through a shared *bus* or *switch*, using *device driver* that presents/abstracts the device in a uniform way

- Typical *interrupt-driven* I/O interaction:
  1. Device driver interface is invoked or called
  2. Device driver sends instructions (and possibly data) to the device controller's registers and local buffer
  3. Device controller performs data transfer to/from local buffer, as specified by the instructions
  4. Upon completion, device controller sends an interrupt to the device driver; control returns to the operating system, possibly with new data
- For higher performance, *direct memory access* (DMA) may be implemented: device controller reads/writes main memory directly, without CPU intervention

Computer System Architecture

- The CPUs referred to so far are assumed to be *general-purpose* — they execute an instruction set that can perform virtually any function
- Most computer systems have multiple processors, but many are *specialized*, performing very specific functions such as disk activity, input device management, or graphics processing — these don't count as "CPUs"
- What separates different kinds of architectures is the number of available *general-purpose processors* and how they interact

- Single-processor systems: one general-purpose, multiple special-purpose — very common, wide variety
- Multiprocessor systems: more than one general-purpose processor; varies on how other components are shared (or not), and how the processors interact
  - ◇ *n* processors does ***not*** mean *n* times faster (Amdahl's Law: speedup is proportional to the fraction of a program that *can* be sped up)
  - ◇ Other benefits: economy of scale, reliability (*graceful degradation*, *fault tolerance*)
  - ◇ Approaches to multiprocessing — these are multiprocessing *models*; they may actually be used on the same hardware
    - *Asymmetric* = one master, multiple slaves
    - *Symmetric* = peers coordinated by shared data structures and resources; approached used by most multiprocessor-capable operating systems today
- Clustered systems: multiprocessor but typically more autonomous; asymmetric vs. symmetric also applies

Operating System Structure

Two primary (and related) operating system functions drive their structure

- *Multiprogramming:* the ability to maintain, and switch among, more than one *job* or *process* at a time; for maximum efficiency, switch occurs when the current job has to wait for something, such as I/O
- *Time sharing* or *multitasking:* the ability to switch among jobs quickly enough to give multiple users a perceived *interactive* experience — relies on the fact that "human speed" is significantly less than computer speed

Based on these functions, we derive the common, high-level components of modern operating systems:

- A *process manager* makes decisions about the *jobs* or *processes* in the overall *job pool*, ranging from which jobs should be in main memory (*job scheduling*) to which job is being "serviced" by the CPU(s) (*CPU scheduling*)
- A *memory manager* coordinates what is or isn't in memory; a key memory manager function is to implement a *virtual memory* scheme, since the total required memory frequently exceeds what is available
- The volatility of main memory requires a *file system* that is explicitly responsible for nonvolatile storage
- Finally, as mentioned, *input/output* plays an underlying role in many of these activities

Operating System, Uh, Operations

- Modern, interrupt-driven operating systems are precisely that: interrupt-*driven* — virtually all activity is initiated by the arrival of an interrupt
- There are different *types* of interrupts, each with a corresponding *handler* or *service routine*; an *interrupt vector* is an in-memory map that determines which interrupt leads to which handler
- *Traps* or *exceptions* are *software-generated* interrupts, caused either by processing errors (e.g., division by zero) or a request from a user process

- Thus, at any given time, a processor is executing some instruction from either a *user program* or the *operating system* (via an interrupt)
- Since instructions may modify memory, including other instructions in memory, programs must be *protected* from each other — one bad or buggy program must not be allowed to mess up other programs, ***especially*** the operating system
- Modern processors address this need through *dual-mode operation* — a hardware-level bit that determines whether a processor is currently executing *user-defined* code (*user mode*) vs. *operating system* code (*kernel mode*; also *supervisor*, *system*, or *privileged* mode)

Dual-Mode Operation

- Dual mode uses (1) a hardware-level *mode bit* that determines the current mode, and (2) a designated set of *privileged instructions* — these can *only* be performed in kernel mode, or else a trap occurs
- The system starts up in kernel mode, then switches to user mode when a user application runs
- While a user application is running, interrupts may switch to operating system code (and thus kernel mode); sometimes the user code requests this explicitly, through a special *system call* interrupt

- The system call interrupt switches the processor to kernel mode and provides a parameter that determines what service was requested; the operating system performs this service, switches to user mode, and returns to the user program
- A hardware *timer* is another mechanism, in addition to dual-mode operation, that maintains an operating system's overall control the computer system
    - The timer works by triggering an interrupt after a specified time period; the operating system can (in kernel mode of course) set this period as needed
    - The timer interrupt guarantees a "drop" into the operating system, regardless of a user program's state

Process Management

- As mentioned, a major component of an operating system's work is managing and switching among multiple *processes* — units of work — within a system
- Concordant with dual-mode operation, processes may belong to the operating system or a user
- Processes may create other (sub)processes, such as when an application is opened from a task bar
- Processes may also internally have multiple *threads*, represented at this level by associating multiple *program counters* with the process, one for each thread

- Other aspects of processes necessitate mechanisms for coordinating among them (process synchronization, process communication, and deadlock handling):
  - *Process interaction* — processes may depend on or work with one another
  - *Resource allocation* — processes share finite resources such as memory and devices
- Overall, a process manager's tasks thus include:
  - Creating and deleting user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication
  - Providing mechanisms for deadlock handling

Memory Management

Though technically "just another layer" in the overall storage hierarchy, main memory is special in that it serves as the primary intermediary between the CPU and other devices

- Layers "above" main memory are either internal to the CPU (registers) or provide "speedup stand-ins" for main memory (caches)
- Layers "below" main memory (disks, removable media) are actually I/O devices, and use main memory to deliver their data to the CPU

- The code and data for currently running processes must also reside in main memory — after all, the von Neumann architecture is all about fetch-read-operate-write between CPU and memory — and thus there's seldom enough physical main memory to go around
- Thus, we have *virtual memory* schemes that allow a system to use the lower (but much larger) storage levels to simulate more main memory (*logical*) than there really is (*physical*)
- An operating system has to handle all of this, in a way that is as transparent to its processes as possible:
  - ◇ Tracking who is currently using which sections of memory
  - ◇ Deciding what gets moved into and out of main memory
  - ◇ Allocating and deallocating memory as needed

Storage Management

- The volatility of main memory requires an explicit mechanism for nonvolatile, permanent storage
- Modern operating systems provide *file systems* that represent a logical abstraction of permanent storage; an individual *file* serves as the logical storage unit
    - ◇ Files and file systems provide a uniform model under which a wide variety of devices — magnetic disks, optical disks, flash drives, tapes — may be accessed
    - ◇ Files not only contain stored data but also *metadata* — data about the files themselves — such as relevant dates, their creators, and permissions
    - ◇ File management involves creating, deleting, organizing, and otherwise manipulating files

- The storage level immediately below main memory (typically hard drives) is also referred to as *secondary storage* — as mentioned, it is used not only for storing user programs and data, but also for system-level functions such as virtual memory
- Multiple physical drives may be combined into a single logical drive (e.g., RAID) — this can be handled either by a specialized controller or in software by the operating system
- Finally, removable media (tapes, CDs, DVDs) come "in and out" of the computer system, and are referred to as *tertiary storage* — they are also presented by the operating system to the user as file systems, and must also have their own device controllers and drivers

Caching

- Recall that the storage hierarchy has the fastest but smallest memory at the top, getting larger and slower (but cheaper) as you travel "down" the hierarchy
- While "cache" may refer to one or more storage layers above main memory, *caching* in general is the temporary copying of data across levels to gain speed
- Depending on what levels are involved, *cache management* may be handled by hardware (e.g., main memory » cache) or by the operating system (e.g. disk » main memory — a "disk cache")

I/O Subsystem

- The *I/O subsystem* is the framework under which device drivers "present" their corresponding device controllers the operating system, which are then exposed uniformly to other programs
- The I/O subsystem interacts with the memory manager when buffering, caching, and spooling data to and from attached devices
- Ideally, programs see a generalized, uniform device driver interface, with individual device specifics and peculiarities encapsulated within the actual drivers

Protection and Security

- We have already seen a number of *protection mechanisms* — anything that controls which processes or users have access to which resources: dual-mode operation, timers, file permissions, driver abstraction
- Today's multiuser and highly networked environment necessitates additional levels of *security* beyond these fundamentals; security can be viewed as a *defense* against possible intentional or malicious attacks
  - ◇ Viruses, trojan horses, worms
  - ◇ Denial-of-service, theft-of-service
  - ◇ Identity theft

Basic security mechanisms include:

- *User* or *security identifiers* to uniquely distinguish among a computer system's users; a user must *authenticate* himself or herself before further access is granted, and all activities are explicitly associated with a user ID
- *Group identifiers* allow distinction among related *sets* of users instead of individuals
- An explicit *privilege escalation* mechanism facilitates proper monitoring and tracking of user activities that require additional permissions

Security concerns were once viewed as external to an operating system, but many now view security as part of an operating system's core functionality

Distributed, Special-Purpose, and Other Systems

Additional variations to how computers are used and organized expand operating system duties even further:

- Communication across two or more systems — a *network* — requires *protocols* above the standard I/O device driver abstraction
- A *network operating system* expands the usual functions of an operating system (particularly process and file management) across computers on a network
- A *distributed operating system* attempts to make multiple networked computers behave like a single one

- *Embedded systems* are computers in specialized devices such as car engines, TVs, DVD players, or appliances — they're still computers, so they have operating systems
- Embedded systems frequently need *real-time operating systems*, where proper operation depends on rigid time constraints — actions must not only be performed correctly, but also within a very specific amount of time
- *Multimedia applications* manipulate storage units that are sufficiently large to merit new approaches to operating system functions (processes, memory, I/O, etc.)
- *Handheld devices* go the other way, significantly constraining resources such as memory, CPU speed, and display size — all of which affect operating system design and implementation choices

Computing Environments

- A final factor that determines and drives operating system functionality is the evolving *environments* within which computers are used
- In the mainframe era, computing resources were scarce and limited, resulting in batch processing followed by time-sharing
- Early "personal computing" had single-user systems with a high degree of interactivity
- "Office environments" spawned local-area networks with file and print servers

- *Client-server* environments separate large-scale, shared operations (databases, mass storage) from the interactive user interfaces that accessed these services
- *Peer-to-peer* environments, on the other hand, view individual computers as both requiring and providing services ("servents")
- Finally, the Internet in general and the Web in particular have blurred these environments into a world of universally interconnected devices, potentially interacting with each other at different levels
    - ◇ Security concerns have made firewalls commonplace and virus protection a matter of course; encrypted communication is now a necessity, not a luxury
    - ◇ Virtually everything needs to be "Web-enabled" in some sense, creating new classes of servers on one hand and new types of client devices on the other