OS Concepts 1.1: What Operating Systems Do

- **What is an OS?:** OS's vary widely in design and in function, but basically an OS is the software that sits between application programs and computer hardware. It provides an environment in which application programs are executed, by allocating physical resources like CPU, memory, and I/O devices.

OS Concepts 1.2: Computer-System Organization

1.2.1: Interrupts

- **Device Controller:** The I/O managing processor within a device.
- **Device Driver:** A component in the OS that understands how to communicate with its respective device controller and manages I/O to those devices.
- **Interrupts:** Interrupts are used in OS's to handle asynchronous events originating from outside the processor (interrupts originating from within the processor are called exceptions). Device controllers and hardware faults raise interrupts. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is necessary for good system performance.
- **Interrupt Vector:** A table of pointers stored in low memory that holds the addresses of the interrupt service routines.
- **Basic Interrupt Implementation:** The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction. When the CPU detects a device controller has asserted a signal on the wire, it reads the interrupt number and jumps to the respective interrupt-handler routine by using the interrupt number as an index into the interrupt vector. It then saves the current state of whatever was interrupted, and starts execution of the interrupt-handler routine. Once the handler is finished executing, it performs a state restore and returns the CPU to the execution state prior to the interrupt.
- **Interrupt Terminology:** We say that the device controller **raises** an interrupt by asserting a signal on the interrupt request line, the CPU **catches** the interrupt and **dispatches** it to the interrupt handler, and the handler **clears** the interrupt by servicing the device.
- **More Sophisticated Interrupt Implementation:** We need the ability for the following:
  - Defer interrupt handling during critical processing
  - Efficiently dispatch to the correct interrupt-handler without having to first poll all devices to see which one raised the interrupt
  - Multilevel interrupts, so that the OS can distinguish between high and low priority interrupts and respond with the appropriate level of urgency

  - A way for an instruction to get the OS's attention directly (separately from I.O requests), for activities such as page faults and errors such as division by zero. This task is accomplished by "traps".

To do this, most CPU's have two interrupt request lines: one is the nonmaskable interrupt, which is used for events such as unrecoverable memory errors, and the second is the maskable interrupt, which the CPU can turn off before the execution of critical instruction sequences that must not be interrupted. Device controllers use the maskable interrupt to request service.

1.2.2: Storage Structure

- **Firmware:** Software stored in ROM or EEPROM for booting the system and managing low level hardware.

1.2.3: I/O Structure

- **Direct Memory Access (DMA):** Interrupt-driven I/O as described in section 1.2.1 is fine for moving small amounts of data but can produce high overhead when used for bulk data movement, like when moving data to and from nonvolatile memory. DMA is used to avoid this overhead. The device controller sets up buffers, pointers, and counters for its I/O device, and transfers entire blocks of data to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. The CPU is able to perform other work while the device controller is performing these operations.

OS Concepts 1.3: Computer-System Architecture

1.3.1: Single-Processor Systems

- **CPU:** The hardware that executes instructions.
- **Processor:** A physical chip that contains one or more CPU's.
- **CPU Core:** The core is the component of the CPU that executes instructions and contains registers for storing data locally.
- **Single-Processor System:** A computer system with a single processor containing one CPU with a single processing core. These systems often also have other special-purpose processors as well, such as disk, keyboard, and graphics controllers. These special-purpose processors run a limited instruction set and do not run processes; their use is incredibly common and does not turn a single-processor system into a multiprocessor system.

1.3.2: Multiprocessor Systems

- **Multiprocessor Systems:** A computer system containing multiple processors. Traditionally contains two or more processors, each with a single-core CPU.

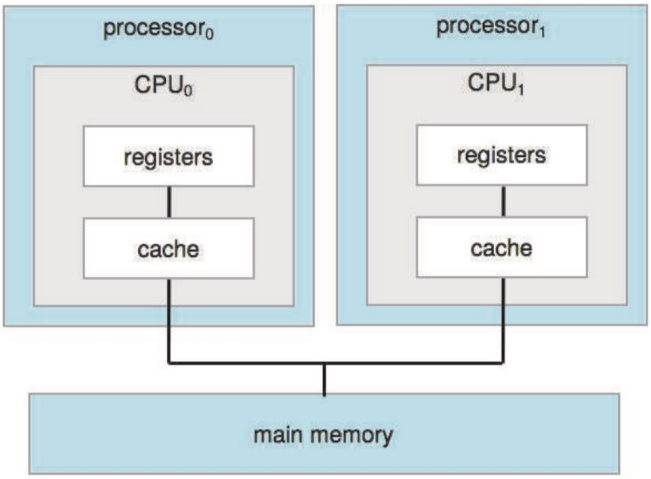

Figure 1: Symmetric *multiprocessing* architecture

- **Multiprocessor Advantages (Increased Throughput):** Primary advantages of multiprocessor systems is increased throughput. The speed-up ratio with N processors is not N, however; it is less than N because there is overhead incurred and contention for shared resources when dealing with multiple processors.

- **Multicore Systems:** A computer system containing multiple cores on the same processor chip. Such systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. Additionally, one chip with multiple cores uses significantly less power than multiple single-core chips, an issue especially important for mobile devices.

- **Multiprocessor Bottleneck:** Adding additional CPU's to a multiprocessor system increases computing power, but does not scale very well. Once too many CPU's are added, contention for the system bus becomes a bottleneck and performance begins to degrade.

- **Non-uniform Memory Access (NUMA):** To avoid bottleneck performance degradation arising from system bus contention, we can provide each CPU with its own local memory that is accessed via a small and fast local bus. The CPU's are connected by a shared system interconnect, so that all CPU's share one physical address space. The advantage is that when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added.

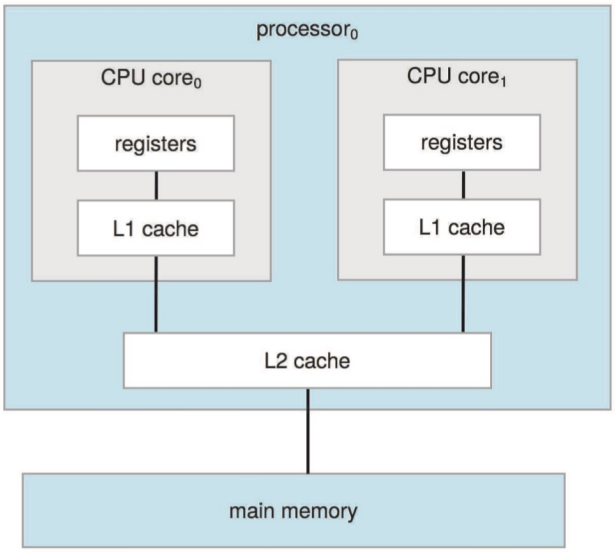

Figure 2: *Multicore* architecture

- **NUMA Drawbacks (Increased Latency):** A potential drawback is increased latency when a CPU must access remote memory across the system interconnect (accessing the local memory of another CPU). OS's can minimize this NUMA penalty through careful CPU scheduling and memory management.

OS Concepts 1.4: Operating-System Operations

- **Bootstrap Program:** The initial program that is run when a computer starts running for the first time. Typically a very simple program and stored in firmware. The program must know how to load the OS kernel into memory and start executing it.
- **System Daemons:** Services provided outside of the kernel that are loaded into memory at boot time, which run the entire time the kernel is running.

1.4.1: Multiprogramming and Multitasking

- **Multiprogramming:** Increase CPU utilization by organizing programs so that the CPU always has one to execute. Execute a process until it needs to wait on some task like I/O, then switch to another process. Keep switching between processes such that the CPU is never idle.
- **Process:** In a multiprogrammed system, a program in execution is termed a process.
- **Multitasking:** The logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them incredibly

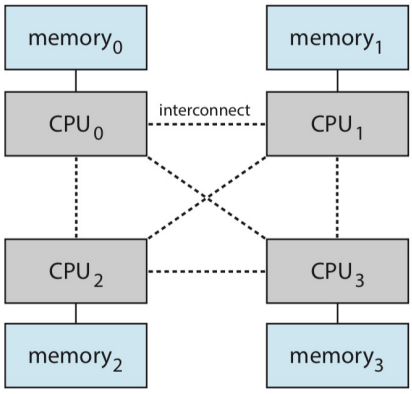

Figure 3: NUMA architecture

frequently. This is how interactive I/O like keyboard input works: rather than letting the CPU sit idle in the time between the keystrokes of the user, the OS will rapidly switch to another process in the meantime.

1.4.2: Dual-Mode and Multimode Operation

- **Modes of Execution:** A properly designed OS must ensure that an incorrect (or malicious) program cannot cause other programs - or the OS itself - to execute incorrectly. To do this, we need to distinguish between the execution of OS code and user-defined code. Most computer systems provide hardware support that allows differentiation among various modes of execution.
- **User Mode and Kernel Mode:** At the very least, we need two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). Whenever the OS gains control of the computer, it is in kernel mode. The system always switches back to user mode before passing control to a user program. The concept of modes can be extended beyond two modes (e.g. the four protection rings in Intel processors).
- **Privileged Instructions:** Some machine instructions that may cause can only be executed in kernel mode. The instruction to switch to kernel mode is an example of a privileged instruction.

1.4.3: Timer

- **Timer:** We must ensure that the OS maintains control over the CPU: we allow user programs to execute, but they must eventually relinquish control to the OS (avoid situations like user program infinite loops or not calling system services and thus not returning control to the OS). To accomplish this, we can use a timer that is set to raise an interrupt after a specified amount of time. If the timer interrupts, control transfers automatically to the OS, which may treat the interrupt as a fatal

error or may give the program more time. Instructions that modify the timer are clearly privileged.

OS Concepts 1.5: Resource Management

The OS can be seen as a resource manager. The following are things that the OS must carefully manage in a computer system.

1.5.1: Process Management

- **Program vs Process:** A program by itself is not a process. A program is a *passive* entity, whereas a process is an *active* entity. Remember that a process is just a program in execution, thus there can be multiple processes associated with the same program (and each is considered a separate execution sequence).

1.5.2: Memory Management

- **Memory Management:** The OS is responsible for keeping tack of which parts of memory are currently being used and which process is using them, allocating and deallocating memory, and deciding which processes (or parts of processes) and data to move into and out of memory.

1.5.3: File-System Management

- **File System:** The OS abstracts the physical properties of its storage devices to define a logical storage unit, the **file**. In other words, the OS implements the abstract concept of a file by managing mass storage media and the devices that control them.

1.5.4: Mass-Storage Management

- **Secondary Storage Management:** The proper management of secondary storage is critical to a computer system. The OS must take care of things such as mounting and unmounting, free-space management, storage allocation, disk scheduling, partitioning, and protection.

1.5.5: Cache Management

- **OS and Memory Hierarchy:** The OS is responsible for moving data between the different levels of the memory hierarchy that it has access to. The OS can only manipulate software-controlled caches, for instance transfer of data from disk to memory, while data transfer from CPU cache to registers is a hardware function.

- **Caches and Multitasking:** In a computing environment where only one process executes at a time, having the same data appear in multiple levels of the memory

hierarchy is not an issue, since access to desired memory always will be to the copy at the highest level of the hierarchy. In a multitasking environment, however, extreme care must be taken to ensure that if several processes wish to access the same data, each of these processes obtains the most recently updated value of the data.

- **Caches and Multiprocessor Systems:** In a multiprocessing environment, not only do we need to make sure that processes access the most recenly updated value of the desired data (multitaking), but we now have CPUs that contain local caches in which data may esist simultaneously in several of these. We need to make sure that an update to the value of a given piece of data in one cache is immediately reflected in all other caches where this data resides. This issue if called *cache conherency*, and is usually handled in hardware (below the OS level).

1.5.6: I/O System Management

- **I/O Subsystem:** One of the purposes of an OS is to hide the peculiarities of specific hardware devices from the user. Often, these peculiarities are hidden from most of the OS itself by the I/O subsystem. Device drivers for specific hardware devices for instance are included in the I/O subsystem.

OS Concepts 1.7: Virtualization

- **Virtualization:** Virtualization allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, etc.) into several different execution environments, creating the illusion that each separate environment is running on its own private computer. An OS that is natively compiled for a particular CPU architecture runs within another OS also native to that CPU.

- **Emulation:** Simulates computer hardware in software, typically used when the source CPU type is different from the target CPU type (e.g. Apple Rosetta when moving from PowerPC to x86). Usually much slower than native code.

OS Concepts 1.10: Computing Environments

1.10.4: Peer-to-Peer Computing

- **Peer-to-Peer (P2P) Computing:** In this distributed computing model, clients and servers are not distinguished from each other. Each node in the system may act as either a client or a server, depending on whether it is requesting or providing a service. In traditional client-server systems, the server is a bottleneck; but in a P2P system, services can be provided by several nodes distributed throughout the network.

- **Centralized P2P:** When a node first joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- **Decentralized P2P:** A decentralized system uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.

OS Concepts 2.6: Why Applications are OS Specific

- **Why Applications are OS Specific:** Each OS exposes different functionalities (system calls), so applications cannot expect to be able to use the same functions across varying OS's. Even if system calls were somehow uniform, other barriers would still pose a challenge: binary formats, varying CPU ISA's, system call discrepencies (specific operands and operand ordering, how to invoke syscalls, syscall result meanings, etc.).
- **How to Make Applications Cross Compatible Across OS's:**
  - Write the application in an *interpreted language* (e.g. Python or Ruby); performance typically suffers. and interpreter usually only offers a subset of the OS's features.
  - Write the application in a language that includes a *virtual machine* (e.g. Java). The JVM has been ported to many OS's, and in theory any Java app can run within the JVM wherever it's available. Usually have similar disadvantages as with interpreted languages.
  - Use a language that compiles *machine and OS specific binaries* (e.g. C++ or Rust), and simply port to each OS on which it will run. Standard API's like POSIX can make this process easier.

OS Concepts 2.7: OS Design and Implementation

2.7.1: Design Goals

- **OS Goals and Specifications:** The first problem in designing a system is to define goals and specifications. These requirements can, however, be divided into two basic groups: user goals and system goals.

2.7.2: Mechanisms and Policies

- **Mechanisms and Policies:** An important principle is the separation of policy from mechanism. Mechanisms determine *how* to do something; policies determine *what* will be done. For instance, the standard Linux kernel has a specific CPU scheduling algorithm, which is a mechanism that supports a certain policy. However, anyone is free to modify or replace the scheduler to support a different policy.

OS Concepts 2.8: OS Structure

2.8.1: Monolithic Structure

- **Monolithic Structure:** A monolithic kernel places all of the functionality of the kernel into a single, static binary file that runs in a single address space. Everything below the system-call interface and above the physical hardware is the kernel, as seen in figure 4. Typically difficult to implement and extend, but have good performance due to very little overhead in the syscall interface, and communication within the kernel is fast.

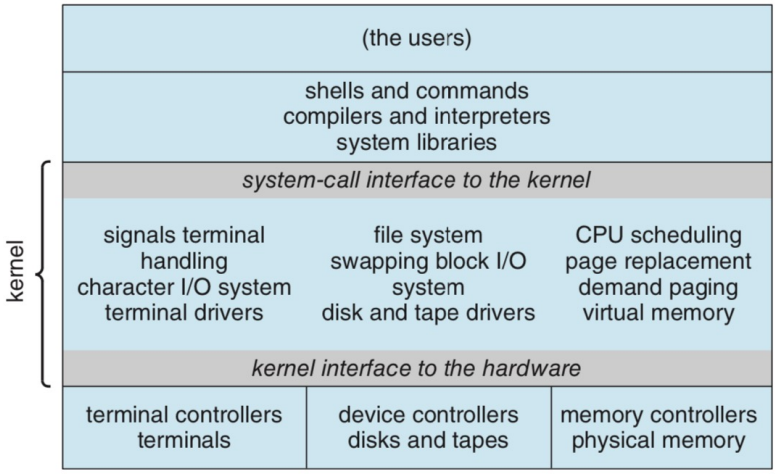

Figure 4: Traditional UNIX system structure (monolithic architecture)

2.8.3: Microkernels

- **What is a Microkernel:** The microkernel approach structures the OS by removing all nonessential components from the kernel and implementing them as user-level programs that reside in separate address spaces, resulting in a smaller kernel, seen in figure 5. There is little consensus on what services remain in the kernel, however, typically minimal process and memory management and a communication facility are provided. The main function of the microkernel is to provide communication between the client program and the various services also running in user space; communication is provided though message passing.

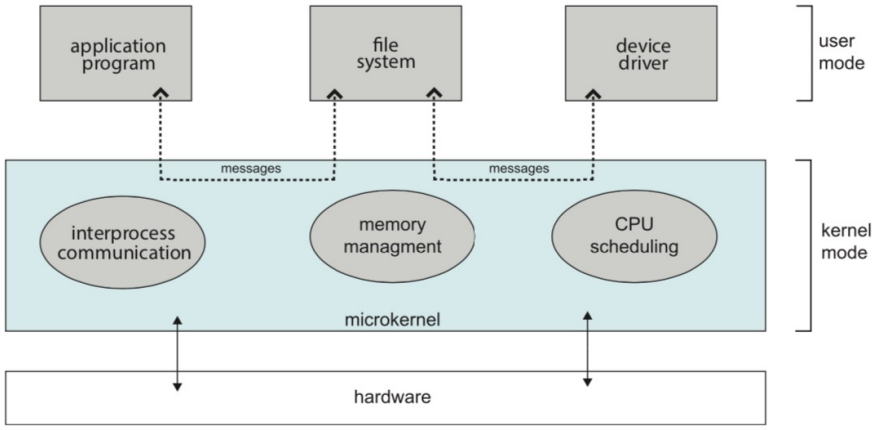

Figure 5: Architecture of a typical microkernel

- **Microkernel Benefits:** The microkernel approach makes extending the OS easier, as all new services are added to user space and do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer as the kernel is smaller. It also makes porting the OS to different hardware easier and provides more security and reliability, as most services are running as user -rather than kernel- processes.
- **Microkernel Drawbacks:** Performance of microkernels can suffer due to increased system-function overhead. The overhead involved in copying messages and switching between processes has been the larged impediment to the groth of microkernel-based OS's.

2.8.4: Modules

- **Loadable Kernel Modules (LKMs):** Using LKMs, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. The key idea is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running.

OS Concepts 2.9: Building and Booting an OS

2.9.2: System Boot

- **Booting OS:** When starting a computer system, how does the hardware know where the kernel is or how to load that kernel? This process of loading the kernel is known as booting the system. The boot process typically roughly follows as so:
    1. A small piece of code known as the bootstrap program or boot loader locates the kernel
    2. The kernel is loaded into memory and started

3. The kernel initializes the hardware
4. The root file system is mounted

- **BIOS:** Some (typically older) computers use a multistage boot process: when the computer first powered on, a small boot loader located in nonvolatile firmware known as BIOS is run. This initial bootloader usually does nothing more than load a second boot lader, which is located at a fixed disk location called the boot block, which is then responsible for loading the OS into memory and begin execution.
- **UEFI:** More recent computers have replaced the BIOS-based boot process with UEFI (Unified Extensible Firmware Interface). The biggest difference is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

OS Concepts 12.1: I/O Overview

- **Accomodating Varying I/O Devices using Device Drivers:** Hardware I/O devices are constantly evolving, and so are methods required to communicate with them. To accomodate this ever increasing variety of I/O devices, the kernel of an OS is structured to use device-driver modules. The device drivers present a uniform device-access interface to the I/O subsystem, much as syscalls provide a standard interface between the application and the OS.

OS Concepts 12.2: I/O Hardware

- **Port:** A device communicates with a computer system via a port.
- **Bus:** If devices share a common set of wires, the connection is called a bus. Uses a rigidly defined protocol that specifies a set of messages that can be sent on the wires. Historically called a data highway. Widely used in computer architecture and vary in signaling methods, speed, throughput, and connection methods.
- **Controller:** A controller is a collection of electronics that can operate a port, a bus, or a devices.

12.2.1: Memory-Mapped I/O

- **Processor and Device Communication:** A controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. I/O device control typically consists of four registers: the status, control, data-in, and data-out registers. Data registers are typically 1 to 4 bytes in size, meaning the processor needs to react quickly otherwise new data may overflow the registers and overwrite the old data. Also means I/O is basically done byte by byte.

- **Memory-Mapped I/O:** With memory-mapped I/O, the controller registers are mapped into the address space of the processor. The CPU executes I/O instructions using standard memory data transfer instructions to read and write the controller registers at their mapped locations in physical memory.
- **Memory-Mapped I/O Example:** Memory-mapped I/O for graphics. A thread sends output to the screen by writing data into the memory-mapped region. The controller then generates the screen image based on the contents of this memory. Writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions to read and write byte by byte.

12.2.2: Polling

- **Polling:** Polled mode I/O between a host and a controller is basically where the host has to repeatedly loop to check if the device is ready to read or write by continually checking the controller status register. Polling the controller naturally requires CPU cycles (read a device register, logical-and to extract a status bit, and branch depending on status). Polling becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service, while other useful CPU processing remains undone.

12.2.3: Interrupts

- **Purpose of Interrupts:** Interrupts are used throughout modern OS's to handle asynchronous events and to trap to supervisor-mode (kernel mode) routines.
- **Interrupt Handlers and OS Boot Time:** At boot time, the OS probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector.
- **Software Interrupts (Traps):** To get the attention of the OS, a special instruction called a trap can be executed. This instruction has an operand that identifies the desired kernel service. Library functions to issue syscalls typically implemented using traps.
- **Interrupt-driven I/O vs Polled I/O:** Interrupt-driven I/O is now much more common than polling, with polling being used for high-throughput I/O. Some device drivers use both: interrupts when the I/O rate is low, and switch to polling when the rate increases to the point where polling is faster and more efficient.

12.2.4: Direct Memory Access

- **Direct Memory Access (DMA):** Using the expensive general purpose CPU processor to watch status bits and to feed data into a controller register one byte at a time (programmed I/O, PIO) seems wasteful. We can avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor

called a DMA controller. Essentially, DMA allows us to bypass the CPU and utilize direct memory to device I/O.

- **DMA High Level Implementation:** Basically, the CPU gives the DMA controller pointers to the source and destination locations and the number of bytes to be transferred. The DMA controller then proceeds to operate the memory bus directly, allowing us to perform I/O without the help of the CPU. When the entire transfer is finished, the DMA controller interrupts the CPU.
- **Cycle Stealing:** When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its caches. Although this cycle stealing can slow down CPU computation, offloading the data transfer work to a DMA controller generally improves the total system performance.

OS Concepts 12.3: Application I/O Interface

- **Accomodating Varying I/O Devices in an OS:** The wide variety of available devices poses a problem for OS implementors (each device has its own set of capabilities, control-bit definitions, and protocols for interacting with the host). How can the OS be designed so that new devices can be attached to the computer without rewriting the OS? And when devices vary so widely, how can the OS give a convenient, uniform I/O interface to applications? Answer: by abstracting I/O hardware with device drivers.
- **Device Drivers:** Device drivers are kernel modules that internally are custom-tailored to specific devices but that export one of the standard OS I/O interfaces. The purpose of the device driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel. Hardware manufacturers can design new devices to be compatible with existing host controller interfaces (such as SATA), or they can write device drivers for popular OS's. Each OS has its own standards for the device driver interface, so they must be ported for each OS.
- **Device Access Conventions:** I/O devices vary among many dimensions, such as synchronous vs asynchronous, sequential or random access, speed of operation, etc. But for the purpose of application access, many of these differences are hidden by the OS, and the devices are grouped into a few conventional types. The major access conventions include: block I/O, character-stream I/O, memory-mapped file access, and network sockets.

12.3.3: Clocks and Timers

- **Programmable Interval Timer:** Most computers have hardware clocks and timers that provide three basic functions: give the current time, give the elapsed time, and set a timer to trigger operation X at time T. This hardware is called a programmable interval timer. It can be set to wait a certain amount of time and

then generate an interrupt, with the option of generating this interrupt periodically as well. The precision of triggers to generate interrupts is limited by the resolution of the timer, together with the overhead of maintaining virtual clocks.

- **Virtual Clocks:** Used to support more timer requests than the number of timer hardware channels. The kernel implements this simply by having a list of of interrupts scheduled both by the kernel and user requests, sorted in earliest-time-first order. It sets the timer for the earliest time, and when the timer interrupts, the kernel just signals the requester that the timer has gone off and reloads the timer with the next earliest interrupt time.

12.3.4: Nonblocking and Asynchronous I/O

- **Blocking I/O** A blocking call causes the execution of the calling thread to be suspended. Blocking application is easier to write than nonblocking application code.

- **Nonblocking I/O:** Nonblocking calls do not suspend the calling threads execution, like blocking calls do. Instead, it returns quickly, with a return value that indicates how many bytes were transferred. One way an application writer could implement this is with multithreading: some threads can perform blocking system calls, while others continue executing. Some OS's also provide nonblocking syscalls.

- **Asynchronous Calls:** An alternative to nonblocking calls. An asynchronous call returns immediately, without waiting for I/O or whatever computation to complete. The calling thread continues to execute its code. The completion of the task at some future time is then communicated to the thread (either setting some variable in the thread address space, or triggering a signal or software interrupt or a call-back routine that is executed outside the control flow of the thread).

12.3.5: Vectored I/O

- **Vectored I/O:** Allows one syscall to perform multiple I/O operations involving multiple locations. This allows multiple separate buffers to have their contents transferred via one syscall, avoiding context switching and syscall overhead. Some versions also provide atomicity.