Justin Ciocoi

Oct. 27, 2023

CSCI 375 Textbook Notes

Chapter 1: Introduction to Operating Systems

1.1: What Operating Systems Do

- We will begin this discussion by breaking down a computer system into its more rudimentary components
- Specifically, this section will focus on *hardware*, an *operating system*, *application programs*, and the *end user*
- The **hardware** components, CPU, memory, and I/O Devices, provide the basic computing resources needed by the system
- The **application programs** define the different ways in which these resources can be used to solve an end user's computing problems
- An operating system, by itself, performs no useful functions but rather provides an environment in which other programs can do useful work
- **1.1.1: User View**
    - The user view can vary according to the computer experience, whether it is a traditional desktop, laptop, or smartphone.
    - In these cases, the operating system is designed mostly with *ease of use* in mind
- **1.1.2: System View**
    - From the system's point of view, the operating system is the program which is most intimately involved with the system hardware
    - Thus, in this context, the operating system acts as a *resource allocator* for the various computing resources present in system hardware
    - This context also tasks the operating system with acting as a *control program*

- A *control program* refers to a program which manages the execution of user programs to prevent errors and improper resource usage
    - It is especially concerned with operation and management of I/O devices

- **1.1.3: Defining Operating Systems**
    - The term *operating system* is a fairly broad one that encompasses many roles and functions present in a computer system
    - This is because although the history of the computer is relatively short, the technologies used in computing have evolved rapidly due to *Moore's Law*
    - *Moore's Law* refers to the prediction made by the founder of Intel, Gordon Moore, that the number of transistors on an integrated circuit will double roughly every 18 months
        - This idea was later amended to every 24 months, which has largely held true
    - In general, when trying to come to a definition of the term *operating system*, there is no single adequate definition that could describe everything that acts as an operating system
    - Operating systems main reason for existence comes from a need to make a computer system usable for a human being.
        - Thus, the fundamental role of operating systems is to solve that problem by executing programs and making solving user processes easier
    - The operating system does this by implementing most of the common functions found among different processes, and implementing them such that each process can better take advantage of computer resources
    - A more general definition for an operating system is the one program that remains running at all times on the computer
        - This program is usually called the **kernel**
    - Along with the kernel, are the application programs we mentioned earlier, as well as *system programs* which are associated with the operating system while not directly being a part of the kernel
    - In the modern era of personal computing, mobile operating systems include not only a core kernel, but also a variety of *middleware*

        - **Middleware** refers to a set of frameworks that provide additional services to application developers
        - For example, both Apples *iOS* and Google's *Android*, feature a core kernel as well as middleware which supports things like databases, multimedia, or graphics
    - Thus, for our purposes we can define operating systems as a core kernel which is always running, middleware frameworks that ease application development, and system programs that aid in managing the system while it is running

1.2: Computer-System Organization

- A modern *general-purpose* computer system consists of one or more CPUs and a number of chips which act as device controllers connected through a common *bus* which provides access between components and shared memory
- Each device controller is in charge of a specific type of device, whether that be *keyboards, displays*, or *video cards*
- Typically, operating systems have a **device driver** for each device controller which will understand the device controller and provide the rest of the operating system a uniform interface to that type of device
    - To ensure orderly access to the shared memory, a memory controller can synchronize access to that memory
- **1.2.1: Interrupts**
    - In order for a program to start an I/O operation, the device driver loads the appropriate registers in the device controller.
    - The controller will then examine the contents of those registers and determine which action to take
    - The controller begins the transfer of data, and once the transfer is complete, the controller informs the device driver that it has finished, and the driver returns control to other parts of the operating system
    - How can the controller inform the driver that it has finished its operation
        - This can be accomplished using an **interrupt**
    - At any time, hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus

        - While other buses do exist in a computer system, the system bus is the main communication path between components
    - Whenever the CPU is interrupted, it stops what it is doing and transfers execution to a fixed location
        - This fixed location generally contains the starting address where the service routine for the interrupt is located
        - The interrupt service routine executes, and upon its completion, the CPU is free to resume the interrupted computation
    - Each computer design has its own interrupt mechanism, but there are several functions which are common among them
    - Since interrupts must be handled quickly, a table of pointers is used to point to interrupt routines.
        - The interrupt routine can be called indirectly through the table with no intermediate routine needed
        - Generally, this table is stored in *low memory*, meaning the first few hundred memory locations
    - The interrupt architecture should also explicitly save the current state of the process it is interrupting such that the interrupted process can later resume as if it had never been interrupted
    - The basic interrupt mechanism works by utilizing a CPU hardware feature known as the *interrupt-request line*, which is a wire that the CPU will sense after executing every instruction
    - When the controller senses that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and uses the table of pointers to jump to the appropriate interrupt routine
    - We say that the device controller **raises** an interrupt by asserting a signal on an interrupt-request line, the CPU **catches** the interrupt, **dispatches** it to the interrupt handler, and allows the handler to **clear** the interrupt by servicing the device
    - Here is a visual representation of what is described above
    -

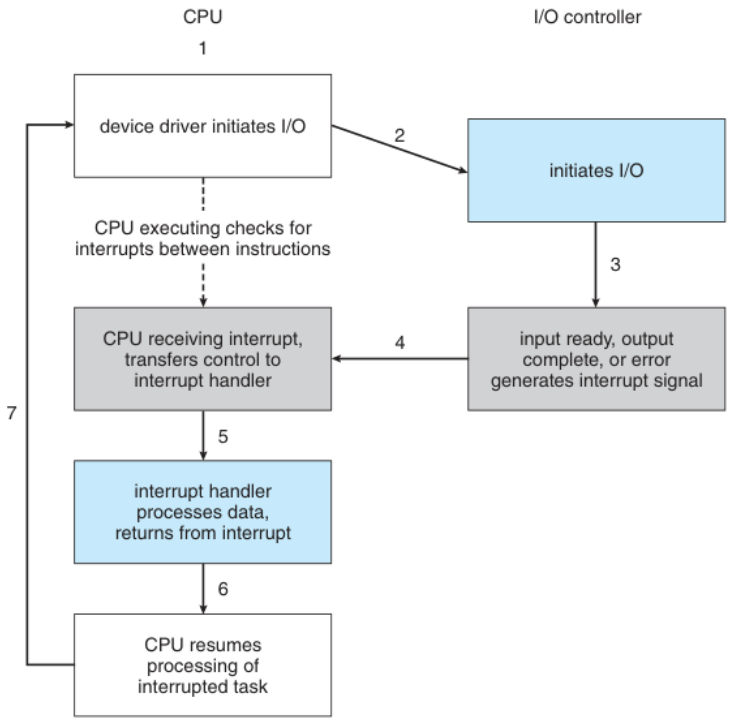


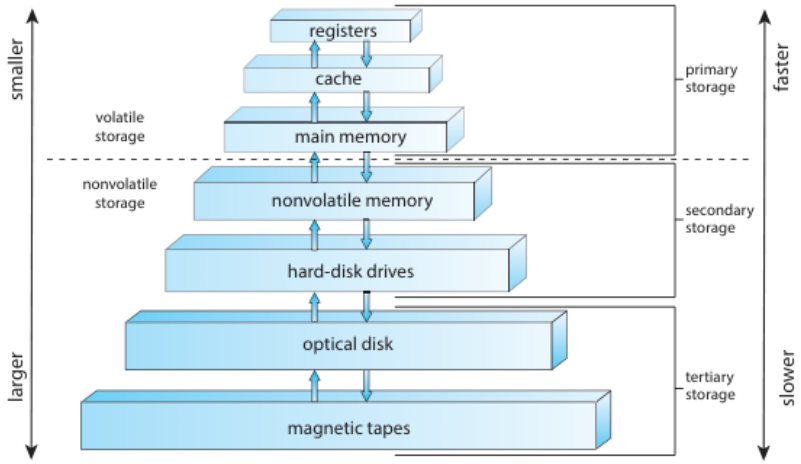
Figure 1.4 Interrupt-driven I/O cycle.

- This had been a fairly rudimentary explanation of interrupt architecture, but in modern operating systems we will need more sophisticated interrupt handling features
  - Interrupt handling must be able to be deferred during critical processing
  - Dispatching to the proper interrupt handler must be done efficiently
  - Multilevel interrupts are required such that the operating system can distinguish between higher and lower priority interrupts and respond with the appropriate degree of urgency

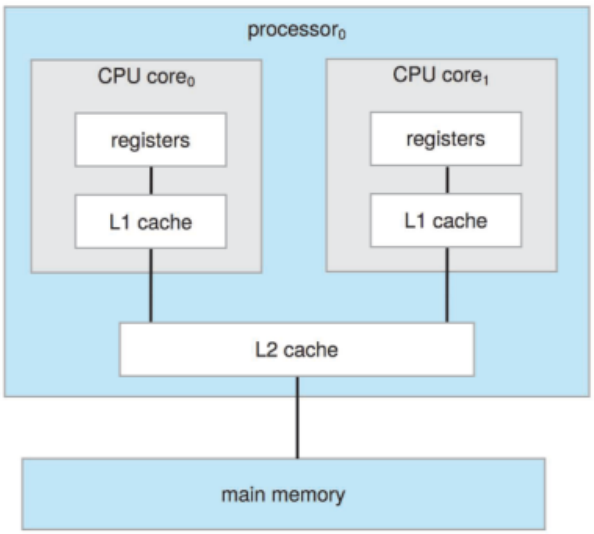
- **1.2.2: Storage Structure**
  - The CPU in a system can only load instructions from memory, so any programs must first be loaded into memory in order to run
  - General purpose computers run most of their programs from rewritable memory, called main memory, and also known as random access memory, or **RAM**
  - Computers also use other forms of memory
    - For instance, since the main memory is volatile, the first program to run on a computer, the *bootstrap program*, is instead stored in electrically erasable programmable read-only memory, or *EEPROM*, which is infrequently written to and *non-volatile* meaning it will not lose its state if it loses power

        - For example, the Apple iPhone uses EEPROM to store serial numbers and hardware information about the device, which should never change regardless of user activity on the device

- The *Von Neumann architecture* refers to the general computer architecture that separates the CPU from main memory
- A typical instruction-execution cycle, as executed on a system with a Von Neumann architecture, first fetches an instruction from memory and then stores that instruction in the *instruction register*
- The instruction will then be decoded and may cause operands to be fetched from memory and stored in some internal register
- Ideally, all needed data and programs would be permanently held on the main memory, but due to its volatile nature and usually small storage capacity, secondary storage systems are almost always used as an extension of main memory
    - The most common of these are *Hard-Disk Drives (HDDs)* and *Nonvolatile Memory (NVM)* devices, such as *Solid State Drives(SSDs)*
- Most programs and data will be stored in the secondary storage until they are loaded into the memory
- Secondary storage is therefore often both the source and destination of a process's processing
- Tertiary storage, such as optical discs or magnetic tapes also exist and mostly serve as backups for secondary storage on computer systems
- Below is a diagram illustrating a fundamental view of the storage hierarchy present in computer systems
-



1.3: Computer-System Architecture

- **1.3.1: Single-Processor Systems**
    - In the beginning of computing's history, most systems used a single processor containing one CPU with a single processing core
    - The *core* is the component that executes instructions and registers for storing data locally
    - In this architecture, the CPU is the only processor that can perform a general instruction set as well as instructions from processes, whereas other device-specific processors may exist in a single processor system
- **1.3.2: Multiprocessor Systems**
    - Nowadays, machines from mobile devices to data-center servers almost all use a *multiprocessor* architecture
    - Even more recently, *multicore* architecture where multiple CPU cores reside on a single chip has become more popular
        - Since on-chip communication is faster than between-chip communication, this increases throughput even more than a traditional multiprocessing system
    - Many multi core systems also implement a hierarchical cache system that behaves very similarly to the storage hierarchy pictured in the previous section
        - CPU cores will have private L_1 caches which are smaller than shared L_2 caches
        - This architecture is pictured below
    - 

1.4: Operating System Operations

- As mentioned before, the *bootstrap program* is the first program that will run on any computer
- In order to initialize the system, the bootstrap program must locate the operating-system kernel and load it into memory
- Some services are provided outside of the kernel by system programs which are loaded into memory at boot time to become *system daemons*, or processes which run the entire time the kernel is running
- Another form of the interrupt which was discussed earlier in this chapter is a *trap* or *exception* which is a software-generated interrupt caused either via an error or a specific request from a user program
    - These requests come in the form of a *system call*
- **1.4.1: Multiprogramming and Multitasking**
    - One of the most important aspects of an operating system is the ability to run multiple programs since a single program cannot usually keep either the CPU or I/O devices busy at all times
    - Therefore, *multiprogramming* or allowing multiple processes to run at once, allows CPU utilization to be maximized
    - In multiprogramming, a system keeps several processes in memory simultaneously, and chooses one to begin executing
        - Eventually, the process might have to wait for something, such as an I/O operation, and instead of allowing the CPU to remain idle while waiting, the CPU can begin to execute a different process
    - *Multitasking* comes as a logical extension of multiprogramming, where the switching between processes occur rapidly, which provides the user with a much faster *response time*
    - We have to also make sure that the operating system maintains control of the CPU and that a user program does not get stuck in an infinite loop
        - For this reason, we accomplish a timer whose duration represents the amount of time a user process will be allowed to execute until control of the CPU is passed back to the operating system