Introduction:

Computers and Software, General System software, Resource abstraction & Sharing, Operating system strategies (Batch, Timesharing, real-time, embedded, etc.) Concept of Multiprogramming Operating system organization, Basic functions, Implementation considerations, Computer organization, bootstrapping the machine, Mobile computers, Multiprocessors and parallel computers, Device Management-Device controllers & Device drivers, I/O strategies (direct I/O with polling, Interrupt driven I/O, DMA), Buffering, Disk scheduling strategies.

System Software:

A system software helps the user and the hardware device to function and interact with each other. Basically, it is a type of software which is used to manage the department of a computer hardware to provide the very basic functionalities that are required by the user. In simple words, we can say that system software works like an intermediatory or a middle layer between the user and the hardware. A system software provides a necessary platform or an environment for the other software to work in. Due to this reason a system software plays an important role in handling the overall computer system. A system software is not just limited to a desktop or a Laptop computer system. It has a broad existence in various digital and electronic devices wherever there is a usage of a computer processor. When a user turns on the computer, it is the system software that gets initialized and gets loaded in the memory of the system. The system software runs in the background and is not used by the end-users. This is the reason why system software is also known as 'low-level software'.

Important features of System Software:

Computer manufacturers usually develop the system software as an integral part of the computer. The primary responsibility of this software is to create an interface between the computer hardware they manufacture and the end user.

System software generally includes the following features:

1. High Speed: System software must be as efficient as possible to provide an effective platform for higher-level software in the computer system.

2. Hard to manipulate: It often requires the use of a programming language, which is more difficult to use than a more intuitive user interface (UI).

3. Written in a low-level computer language: System software must be written in a computer language the central processing unit (CPU) and other computer hardware can read.

4. Close to the system: It connects directly to the hardware that enables the computer to run.

5. Versatile: System software must communicate with both the specialized hardware it runs on and the higher-level application software that often has no direct connection to the hardware it runs on. System software also must support other programs that depend on it as they evolve andchange.

Types of System Software:

System software manages the computer's basic functions, including the disk

operating system, file management utility software and operating system.

Other examples of system software include the following:

Following are the most common examples of a system software:

- **Operating System (OS):**
  It is one of the popularly used System Software throughout the digital arena. It is a collection of software that handles resources and provides general services for the other applications that run over them. Although each Operating System is different based on look and feel as well as functionalities, most of them provide a Graphical User Interface through which a user can manage the files and folders and perform other tasks. Every device, whether a desktop, laptop or mobile phone requires an operating system to provide the basic functionality to it. An Operating System essentially determines how a user interacts with the system; therefore, many users prefer to use one specific OS for their device. There are various types of operating system such as single user, multiuser, embedded, real-time, distributed, mobile, etc. It is important to consider the hardware specifications before choosing an operating system. Some examples of Operating systems software are Microsoft Windows, Linux, Mac OS, Android, iOS, Ubuntu, Unix, etc.
- **Device Drivers:**
  It is a type of software that controls the hardware device which is attached to the system. Hardware devices that need a driver to connect to a system include displays, sound cards, printers, mouse, and hard disks. Further, there are two types of device drivers: User Device Driver and Kernel Device Drivers. Some examples of device drivers area Drivers, VGA Drivers, Virtual Device Drivers, BIOS Driver, Display Drivers, Motherboard Drivers, Printer Drivers, ROM Drivers, Sound card Driver, USB Drivers, USB Drivers, etc.
- **Firmware:**
  It is the permanent software that is embedded into a read-only memory. It is a set of instructions permanently stored on a hardware device. It provides essential information regarding how the device interacts with other hardware. Firmware can be considered as 'semi-permanent' as it remains permanent unless it is updated using a firmware updater. Some examples of firmware are: BIOS, Computer Peripherals, Consumer Applications, Embedded Systems, UEFI, etc.
- **Programming Language Translators:**
  These are mediator programs on which software programs rely to translate high-level language code to simpler machine- level code. Besides simplifying the code, the translators have the capability to Assign data storage, enlist source code as well as program details, offer diagnostic reports, Rectify system errors during the runtime. Examples of Programming Language Translators are Interpreter, Compiler and Assemblers.

- **Utility:**
  This software is designed to aid in analyzing, optimizing, configuring and maintaining a computer system. It supports the computer infrastructure. This software focuses on how an OS functions and then accordingly it decides its trajectory to smoothen the functioning of the system. Software like antiviruses, disk cleanup & management tools, compression tools, defragmenters, etc are all utility tools.

Resource abstraction & Sharing:

Resource abstraction is the process of hiding the details of how the hardware operates, thereby making computer hardware relatively easy for an application programmer to use.

The main purpose of an operating system is to provide an interface between the hardware and the application programs and to manage the various pieces that make up a computer. To be more precise, these pieces are called resource.

A resource is any object which can be allocated within a system.

Some examples of resources are processors (CPUs), input/output devices, files and memory (RAM). In a computer system, abstractions are used to eliminate tedious detail that a programmer otherwise would have to handle. Without a suitable abstraction for writing characters to a screen (such as a print function), we would have to learn to set a screen bitmap so that it would print “Hello, world” in 12 point Arial font on a video display. Rather than learning all those details, the C programmer just learns about printf() and the stdio library. The time that a programmer would have spent writing code to form characters on a screen can now be spent writing code to solve the problem at hand.

Resource abstraction has its tradeoff, however. While making the hardware easier to use, resource abstraction also limits the specific level of control over the hardware by hiding some functionality behind the abstraction. Since most application programmers do not need such a high level of control, the abstraction provided by the operating system is generally very useful.

Operating system strategies:

Batch Processing:

This strategy involves reading a series of jobs (called a batch) into the machine and then executing the programs for each job in the batch. This approach does not allow users to interact with programs while they operate.

To improve utilization, the concept of a batch operating system was developed. It appears that the first batch operating system (and the first OS of any kind) was developed in the mid concept was subsequently refined and implemented on the IBM 704 by a number of IBM customers. By the early 1960s, a number of vendors had developed batch operating systems for their computer systems. IBSYS, the IBM operating system for the7090/7094 computers, is particularly notable because of its widespread influence on other systems. In a batch processing operating system, the prime concern is CPU efficiency. The batch processing system operates in a strict one job-at-a-time manner; within a job, it executes the programs one after another. Thus only one program is under execution at any time. The opportunity to enhance CPU efficiency is limited to efficiently initiating the next program when one program ends, and the next job when one job ends, so that the CPU does not remain idle.

Time sharing:

- In Time sharing (or multitasking) systems, a single CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. The user feels that all the programs are being executed at the same time.
- Time sharing requires an interactive (or hands-on) computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard o a mouse, and waits for immediate results on an output device. Accordingly, the response time should be short— typically less than one second.
- A time-shared operating system allows many users to share the computer simultaneously. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use only, even though it is being shared among many users.
- A multiprocessor system is a computer system having two or more CPUs within a single computer system, each sharing main memory and peripherals. Multiple programs are executed by multiple processors parallel.

Time sharing requires an interactive computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the response time should be short — typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is

given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device. Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

Real-Time Embedded Systems:

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to DVDs and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems — such as Linux — with special-purpose applications to implement the functionality. Others are hard- ware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices with application- specific integrated circuits (ASICs) that perform their tasks without an operat- ing system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as elements of networks and the web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer— either a general-purpose computer or an embedded system— can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator can notify the grocery store when it notices the milk is gone.

Embedded systems almost always run real-time operating systems. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real- time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt after it had smashed into the car it was building. A real-time system

functions correctly only if it returns the correct result within its time constraints. Contrast this system with a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or a batch system, which may have no time constraints at all.

Functions provided by the Operating System:(helpful to the user)

1. **User interface**. Almost all operating systems have a user interface (UI). This interface can take several forms. One is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a batch interface, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.
2. **Program execution.** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
3. **I/O operations**. A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
4. **File-system manipulation.** The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.
5. **Communications.** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in which packets of information in predefined formats are moved between processes by the operating system.
6. **Error detection.** The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At

other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct**.**

Functions provided by the Operating System:(for efficient operation)

1. Resource allocation. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.

2. Accounting. We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

3. Protection and security. The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate

Implementation considerations:

Because operating systems are collections of many programs, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.

Early operating systems were written in assembly language. Now, although some operating systems are still written in assembly language, most are written in a higher-level language such as C or an even higher-level language such as C++. Actually, an operating system can be written in more than one language. The lowest levels of the kernel might be assembly language. Higher-level routines might be in C, and system programs might be in C or C++, in interpreted scripting languages like PERL or Python, or in shell scripts. In fact, a given Linux distribution probably includes programs written in all of those languages.

The first system that was not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in the system programming language PL/1. The Linux and Windows operating system kernels are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

The advantages of using a higher-level language, or at least a systems- implementation language, for implementing operating systems are the same as those gained when the language is used for application programs: the code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to port —to move to some other hardware— if it is written in a higher-level language. For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it runs natively only on the Intel X86 family of CPUs. (Note that although MS-DOS runs natively only on Intel X86, emulators of the X86 instruction set allow the operating system to run on other CPUs— but more slowly, and with higher resource use. As we mentioned in Chapter 1, emulators are programs that duplicate the functionality of one system on another system.) The Linux operating system, in contrast, is written mostly in C and is available natively on a number of different CPUs, including Intel X86, Oracle SPARC, and IBMPowerPC.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. This, however, is no longer a major issue in today's systems. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units that can handle the details of complex dependencies much more easily than can the human mind.

Bootstrapping the machine:

We all know that the operating system is stored on Hard disk/SSD. It must be loaded into main memory. The procedure of starting a computer by loading the kernel is known as booting the system. On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

When a CPU receives a reset event— for instance, when it is powered up or rebooted— the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems— such as cellular phones, tablets, and game consoles— store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using erasable programmable read-only memory (EPROM), which is read- only except when explicitly given a command to become writable. All forms of ROM are also known as firmware, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available. For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk.

Mobile computers:

Mobile computing refers to computing on handheld smartphones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight. Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing. Over the past few years, however, features on mobile devices have become so rich that the distinction in functionality between, say, a consumer laptop and a tablet computer may be difficult to discern. In fact, we might argue that the features of a contemporary mobile device allow it to provide functionality that is either unavailable or impractical on a desktop or laptop computer.

Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording high-definition video. Accordingly, tremendous growth continues in the wide range of applications that run on such

devices. Many developers are now designing applications that take advantage of the unique features of mobile devices, such as global positioning system (GPS) chips, accelerometers, and gyroscopes. An embedded GPS chip allows a mobile device to use satellites to determine its precise location on earth. That functionality is especially useful in designing applications that provide navigation— for example, telling users which way to walk or drive or perhaps directing them to nearby services, such as restaurants. An accelerometer allows a mobile device to detect its orientation with respect to the ground and to detect certain other forces, such as tilting and shaking. In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device! Perhaps more a practical use of these features is found in augmented-reality applications, which overlay information on a display of the current environment. It is difficult to imagine how equivalent applications could be developed on traditional laptop or desktop computer systems.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 64 GB in storage, it is not uncommon to find 1 TB in storage on a desktop computer. Similarly, because power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.

Two operating systems currently dominate mobile computing: Apple iOS and Google Android. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers**.**

Multiprocessor Systems

Multiprocessor systems (also known as parallel systems or multicore systems) have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices. Multiprocessor systems first appeared prominently appeared in servers and have since migrated to desktop and laptop systems. Recently, multiple processors have appeared on mobile devices such as smartphones and tablet computers.

Multiprocessor systems have three main advantages:

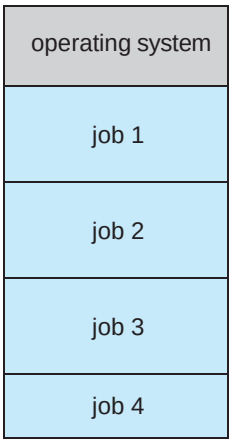
1. **Increased throughput.** By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N, however; rather, it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly, N programmers working closely together do not produce N times the amount of work a single programmer would produce.
2. **Economy of scale.** Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.

3. **Increased reliability.** If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

Multiprogramming:

One of the most important aspects of operating systems is the ability to multiprogram. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memory simultaneously as shown in the figure.



Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory.

The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When that job needs to wait, the CPU switches to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system.
Time sharing (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Device controllers & Device drivers:

Device controllers:

A controller is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a host adapter) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kind of connection — SCSI or Serial Advanced Technology Attachment (SATA), for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device controller can support memory-mapped I/O. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers at their mapped locations in physical memory.

The I/O control level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high- level commands such as “retrieve block 123.” Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller’s memory to tell the controller which device location to act on and what actions to take.

Device Drivers:

A device driver can be thought of as a translator. Its input consists of high- level commands such as “retrieve block 123.” Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller’s memory to tell the controller which device location to act on and what actions to take. The device drivers present a uniform device- access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

For device-hardware manufacturers, each type of operating system has its own standards for the device-driver interface. A given device may ship with multiple device drivers for instance, drivers for Windows, Linux, AIX, and Mac OS X

Polling:

Assume that 2 bits are used to coordinate the producer – consumer relationship between the controller and the host. The host signals its wishes via the command-ready bit in the command register. The host sets the command-ready bit when a command is available for the controller to execute.

For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the
5. busy bit.
6. The controller reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device.

This loop is repeated for each byte. In step 1, the host is busy-waiting or polling: it is in a loop, reading the status register over and over until the busy bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task.

Direct Memory Access (DMA):

While Interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, direct memory access (DMA) is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to 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. While the device controller is performing these operations, the CPU is available to accomplish other work.

Handshaking between the DMA controller and the device controller is performed via a pair of wires called DMA-request and DMA-acknowledge. The device controller places a signal on the DMA-request wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, place the desired address on the memory-address wires, and place a signal on the DMA-acknowledge wire. When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory and removes the DMA-request signal.

When the entire transfer is finished, the DMA controller interrupts the CPU. 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 primary and secondary caches. Although this cycle stealing can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the to tal system performance. Some computer architectures use physical memory addresses for DMA, but others perform direct virtual memory access (DVMA), using virtual addresses that undergo translation to

physical addresses. DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

Buffering:

Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk.

So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation. Since the disk write is not instantaneous and the modem still needs a place to store additional incoming data, two buffers are used. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This double buffering decouples the producer of data from the consumer, thus relaxing timing requirements between them.

A second use of buffering is to provide adaptations for devices that have different data-transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy semantics for application I/O. An example will clarify the meaning of "copy semantics." Suppose that an application has a buffer of data that it wishes to write to disk. It calls the write() system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With copy semantics, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer.

Disk Scheduling:

FCFS- First Come First Served:

(FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37, 122, 14, 124, 65, 67, in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in following Figure:

0 14 37 53 65 67 98 122 124 183 199

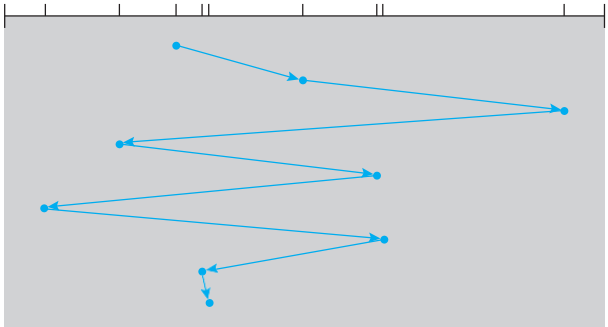


Figure FCFS disk scheduling.

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

SSTF-Shortest Seek Time First:

The SSTF algorithm selects the request with the least seek time from the current head position. In other words, SSTF chooses the pending request closest to the current head position.

Suppose that we have two requests in the queue, for cylinders 14 and 186, and while the request from 14 is being serviced, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.

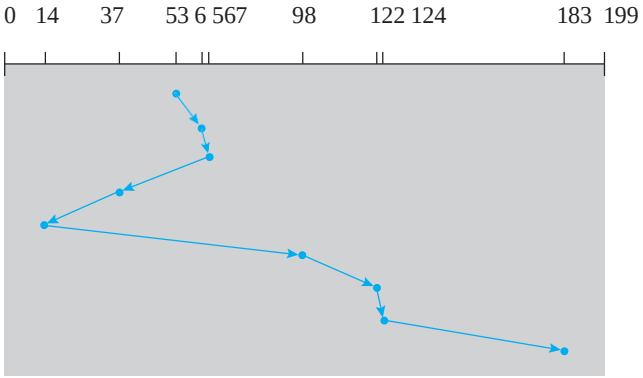


Figure SSTF

SCAN algorithm: SCAN algorithm the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

The SCAN algorithm is sometimes called the elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way. Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183.

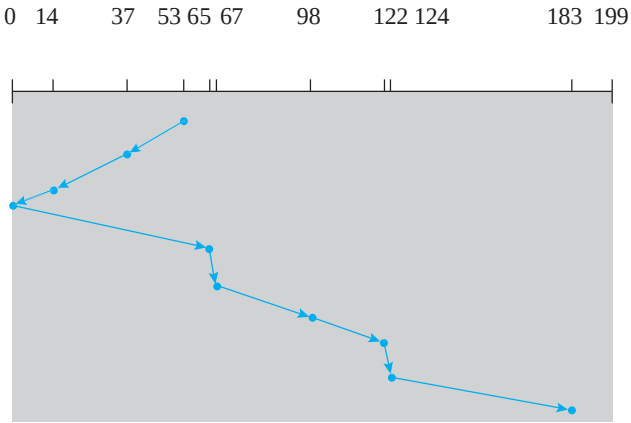


Figure SCAN algorithm

If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

C-SCAN Scheduling:

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

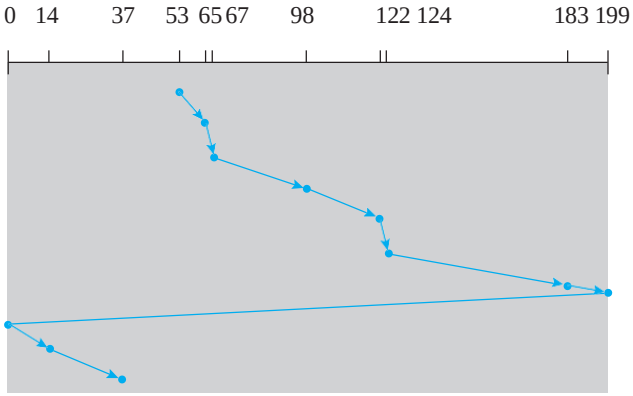


Figure C-SCAN Scheduling

LOOK Scheduling:

As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction.

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

0 14 37 53 65 67 98 122 124 183 199

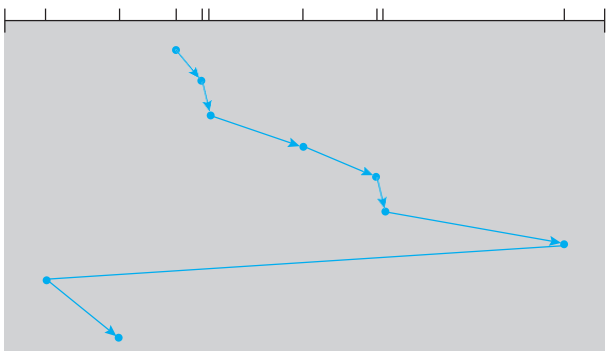


Figure Look Scheduling