Lecture: 2
System Software and Resource Abstraction

System Software:

System software is a type of computer program that is designed to run a computer's hardware and application program i.e. system software is the interface between the hardware and user applications. The operating system is the best known example of system software.

- System software is used to manage the computer itself.
- It runs in the background, maintaining the computer's basic functions so users can run higher-level application software to perform certain tasks.
- Essentially, system software provides a platform for application software to be run on top of.

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 and change.

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:

- **The BIOS (Basic input/output system):** gets the computer system started after it's turned on and manages the data flow between the OS and attached devices, such as the hard drive, video adapter, keyboard, mouse and printer.
- The **boot program** loads the OS into the computer's main memory or random access memory (RAM).
- An **assembler** takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations.
- A **device driver** controls a particular type of device that is attached to your computer such a keyboard or mouse. The driver program converts the more general I/O instruction of the OS to messages that the device type can understand.

System software can also include system utilities, such as the disk defragmenter and System Restore and development tools, such as compilers and debuggers.

System software vs. application software

System software	Application software
General-purpose software that manages basic system resources and processes	Software that performs specific tasks to meet user needs
Written in low-level assembly language or machine code	Written in higher-level languages, such as Python and JavaScript
Must meet specific hardware needs; interacts closely with hardware	Does not take hardware into account and doesn't interact directly with hardware
Installed at the same time as the OS, usually by the manufacturer	User or admin installs software when needed
Runs any time the computer is on	User triggers and stops the program
Works in the background and users don't usually access it	Runs in the foreground and users work directly with the software to perform specific tasks
Runs independently	Needs system software to run
Is necessary for the system to function	Isn't needed for the system to function

Resource Abstraction:

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.

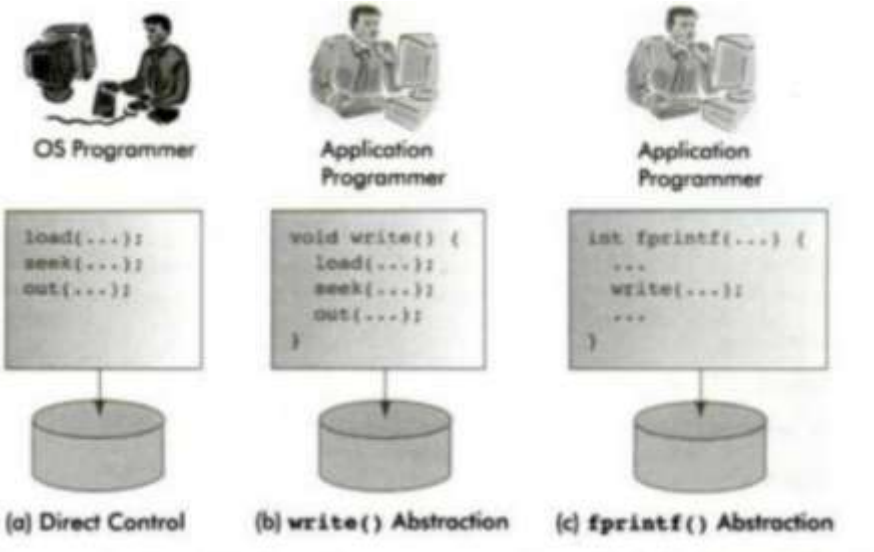
The main purpose of an operating system is to provide an interface between the hardware and the application programs and to manage the various pieces that make up a computer. To be more precise, these pieces are called resource.

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.

Example: An abstraction of a Disk Drive



Let us see how disk output operations can be represented at different levels of abstraction. The device is controlled with software operations for copying a block of information from the computer's main memory into the device's buffer memory.
A series of commands is required to write information from a primary memory block onto a disk such as:

load(block, length, device);
seek(device, 236);
out(device, 9);

A simple abstraction (Fig b) would be to package these commands with other necessary supplementary commands, into a write() procedure such as:

```
void write(char *block, int len, int device, int track, int sector)
{
...
load(block, len, device);
seek(device, 236);
out( device, 9);
...
}
```

Data block addresses on a disk are specified by a track number, such as 236 in the load instruction, and sector number, such as 9 in the out instruction.

A higher-level abstraction might translate every block specification so that a nonnegative integer address is used instead of a disk-specific address such as track 236 in the seek () function and sector 9 in the out() function.

This allows the programmer to ignore physical addresses defined by disk drive technology in favor of logical addresses that apply to any kind of storage device.

Now, an output operation such as:

write(block, 100, device, 236, 9) ;
can be written as
write(block, 100, device, 3788);

An even higher-level abstraction provides software with a way to treat the disk as file storage. Suppose the system software provides a file identification, fileID, as the abstraction of the disk. Then a library such as the C stdio library, can provide a function to write an integer variable, datum (stored in a small memory block), onto the device at an implicit offset from the beginning of the file. The programmer then uses operations such as:

fprintf(fileID, "%id", datum);
to write information to thee disk.