

**Design and Engineering of Computer Systems**  
**Professor Mythili Vutukuru**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Bombay**  
**Lecture - 5**  
**Introduction to Operating Systems**

Hello everyone, welcome to the fifth lecture in the course Design and Engineering of Computer Systems. So, in the previous two lectures, we have studied some basic details about computer hardware. We have studied how the CPU works, what is main memory, how do I/O devices work and so on.

So, if you take a course on computer organization and architecture, you will learn many more details about the hardware. But in this course, we are not going to go into any more details about hardware, because the focus of this course is on the design of real-life systems and we will mostly be covering software aspects of such systems. So, starting from this lecture onwards, we are going to go into the actual contents of the course.

So, in this lecture, we will begin with an introduction to operating systems. So, the next three weeks will go into a lot of detail on operating systems and this lecture is just an overview of what we are going to cover in the next three weeks. So, let us begin.

(Refer Slide Time: 1:16)

**What is an operating system?**

- System software, to manage the computer system hardware
  - Distinct from user software (web browser, web server, gaming engine, ..)
- Special program (compiled OS executable) is stored on hard disk, starts running at boot up
  - Once OS starts, it starts other user programs
- Operating system has kernel + system programs
  - Kernel = the core part of the operating system
  - System programs are useful programs to manage system (e.g., program to list all files in a directory "ls")
- Most OS today (e.g., Linux) are monolithic, one big executable of the kernel
  - Install kernel modules at run time to add extra functionality (e.g., device drivers)
- Alternate architecture: microkernels, more modular but not popular
  - Small core kernel, most functionality as services running on microkernel

Handwritten notes: User Software, OS, H/W, core, services

So, what is an operating system? You all must have interacted with an operating system at some point like Linux or Windows and so, on. So, an operating system is what is called system software that is it is a software that manages the hardware, so that other user software

like a web browser or a server or games or any other apps you have, all of these can run more easily. So, and OS is what is called system software.

So, typically, you have hardware in a computer system and then you have the operating system running on top of the hardware and then all the other user software runs on top of the operating system. So, an operating system is a special program, you can think of it as a special software and this is loaded. This is the first thing that starts when your computer starts up. When your computer boots up this operating system, the executable, that is stored on the hard disk gets loaded into memory and the operating system starts to run.

And from now on the operating system will start other user programs that are required in a computer system and run all of them. So, just some terminology, we keep using the word kernel and operating system interchangeably. So, they are more or less the same. The only difference is that the kernel is basically the core part the central part of an operating system is called a kernel. And an operating system has other small system programs like helper programs also in addition to this kernel.

So, the OS is simply the kernel plus the system programs. So, an example is a program called `ls` that you would have used if you are using a Linux system. If you type `ls`, it will show you all the files in that directory. So, these are all extra system programs that come with your operating system. And most operating systems today are what are called monolithic operating systems. That is the entire operating system is built as one big program, one big executable.

And of course, you can install small things like kernel modules when the system is running for some extra functionality like device drivers and so on, but the core part of the operating system is in the form of one big executable that is loaded when the system boots up. So, there also exists alternate architectures for how operating systems can be built, which is called microkernels, which is the operating system, the core itself is very small. And all the most of the operating system functionality runs as services on top of this core, it is built in a much more modular fashion. But this architecture of operating systems is not very popular today. So, therefore, we will not be covering this in this course.

(Refer Slide Time: 4:14)

### Why do we need operating systems?

- **Convenience:** makes life easy ✓
  - Every user program need not worry about handling hardware on its own
- **Isolation:** makes life secure  $P_1, P_2, \dots, P_n$ 
  - Multiple programs on a computer system are protected from each other
- **Better utilization of resources:** makes life efficient ✓
  - Easier to optimize usage of system resources via careful planning
- Operating systems started out as simple libraries for user programs
  - Later, support for isolation via CPU privilege levels
  - Later, support for multiprogramming

So, before we understand what are operating systems? The question comes up, why do we even need an operating system? If it is not user program that you the user wants to use, then why even bother having this piece of software? So, there are many very important reasons why we need an operating system? The first one is convenience.

So, an operating system makes life easy for you. So, you as a user program do not have to write a lot of extra code, which is handled by the operating system, for example, handling hardware, printing to the display. All of these are common functionalities that every program needs and instead of every program writing it on its own. The operating system provides all of that logic automatically to all user programs.

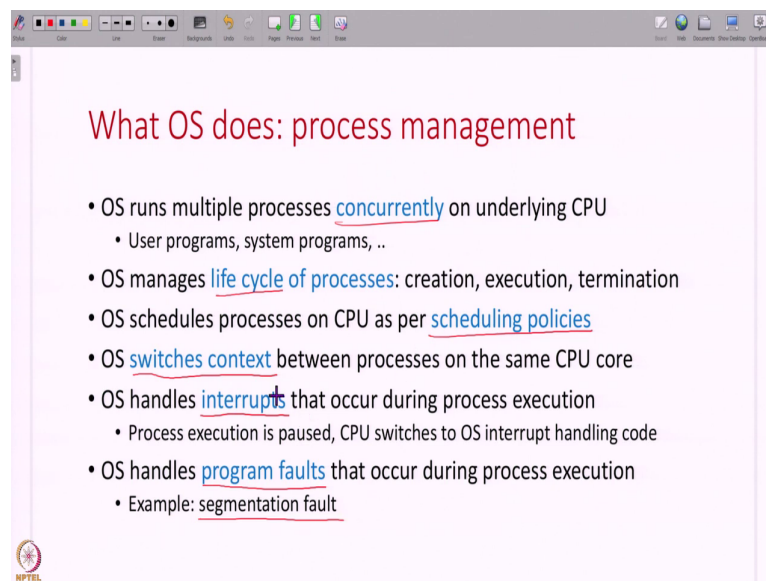
So, in this way using an operating system is very convenient, but that is not all. You also need an operating system for isolation across multiple processes, if you have multiple programs running on the same hardware, all of these programs should be isolated from each other and you cannot trust anyone program to not damage all the other programs. And the way this isolation is achieved, the security is achieved is by using an operating system.

An operating system ensures that the different programs that are sharing the hardware are isolated from each other. And there is also another reason which is like a better utilization of the hardware resources. An operating system brings in efficiency, because you have a central point of contact that is managing all the hardware resources, the OS can do a lot of optimization of these resources via some careful planning and so on as we will see in the course.

So, these are all some of the very good reasons why any user software of a computer system runs on top of a system software like the operating system. So, convenience is of course, as we said the most obvious reason why you would use operating systems therefore operating systems as expected started out as simple libraries for user programs, then of course, they have evolved.

For example, by providing to provide isolation, you need to run at a different CPU privilege level. We have seen these two lectures back that CPUs have multiple privileged levels. So, an operating system runs at a higher privilege level and provides isolation across user programs. So, that functionality got added later, then later on multi programming that is the ability to run multiple programs concurrently got added on. So, operating systems have evolved over the years from simple libraries to providing all of this complex rich functionality that we see today.

(Refer Slide Time: 7:02)



### What OS does: process management

- OS runs multiple processes concurrently on underlying CPU
  - User programs, system programs, ..
- OS manages life cycle of processes: creation, execution, termination
- OS schedules processes on CPU as per scheduling policies
- OS switches context between processes on the same CPU core
- OS handles interrupts that occur during process execution
  - Process execution is paused, CPU switches to OS interrupt handling code
- OS handles program faults that occur during process execution
  - Example: segmentation fault

So, this lecture, I will give you an overview of all the different aspects of operating systems, we will be studying in the coming weeks. The most important thing that an operating system does is process management, it is responsible for concurrently running multiple processes on the underlying CPU. These processes can be programs written by the user, some system programs, that the operating system itself has to run from time to time, all of these programs or processes are run concurrently on the underlying CPU.

And managing the lifecycle of all of these processes. What does lifecycle mean? Creating a process, enabling it to run when a process terminates, cleaning it up all of this lifecycle of

every process is managed by the operating system. So, when you as a user, you write code, you write a program, you do not have to worry about all of these other things, it is taken care of by the operating system.

The other thing that the operating system does is it schedules all of these processes, and how does it decide which process to run if there are twenty processes that want to run on a CPU core, how does it decide which process runs next? So we have what is called scheduling policies, there are certain policies that dictate how processes should be run in what order they should be run on the CPU. So, the scheduling policies are also part of the operating system code.

And another functionality that the OS does is it switches between these processes. So, the scheduling policy decides, next run this process, then run this process and there are scored in the operating system that actually executes that decision. It, we have seen this in previous lecture, where you store the context, you store all the information about the process somewhere, then you start the next process, then you save that context, you restore another context. So, all of this is also managed by the operating system. So, this is called context switching between processes.

And then other things that the OS does is it handles interrupts. Now, we have seen that external devices, whenever they want to talk to the computer system, they will raise an interrupt when they have any event to deliver. So, when this interrupt occurs, the CPU has to stop whatever it is doing to handle the interrupt. So, all of this logic to handle interrupts is also part of the operating system.

We have seen an example of this where the execution of a process is stopped, its context is saved, the operating system runs to handle the interrupt. So, we will study all of this how the operating system exactly does all of this in a lot of detail in this course.

And another thing that they was also does is manage the faults of your program. So, suppose you have written a program which has a bug which is accessing an array out of bounds or something like that it is accessing memory that it is not supposed to. There is some bad behavior in a program.

So, then this program causes a fault and all such faults are also handled by the operating system. If you access an array out of bounds, you may get a segmentation fault. All of such

things are also managed by the operating system. So, overall, everything to do with processes is handled by the operating system.

(Refer Slide Time: 10:21)

**System calls**

- When user program requires a service from OS, it makes a system call
  - Example: Process makes system call to read data from hard disk
  - Why? User process cannot run privileged instructions that access hardware
  - CPU jumps to OS code that implements system call, and returns back to user code
- System calls supported by an OS form the API to user programs
- POSIX API: standard set of system calls defined for portability
  - User program written on one POSIX-compliant OS will run without change on another POSIX-compliant OS
  - However, program may have to be recompiled if architectures are different
- Normally, user program does not call system call directly, but uses language library functions
  - Example: printf is a function in the C library, which in turn invokes the system call to write to screen

Handwritten notes: User code (with arrow), OS2 HW, OS1 HW, API, User → printf → write

So, moving on, I would like to introduce this terminology called a system call. So, what is a system call? Whenever a user program requires any service from the operating system, it makes a system call. For example, if a process wants to read some data that is stored in a file on the hard disk, or it wants to read some input that the user is giving from a keyboard, or your program wants to print something to the monitor, say by using the printf statement, and see.

So, all of these require access to the underlying hardware. And you as a user cannot write code to directly do this, why? Because accessing hardware, all of these require privileged instructions. And user code runs in a non-privileged CPU mode. And you cannot invoke privileged instructions. Therefore, then how do you achieve these tasks in your programs, you will make a system call, you will call the OS and the OS code will run and carry out this functionality for you.

So, when you make a system call, the CPU will jump to the operating system code and a high privilege level, it will implement the system call, it will execute the system call and come back to user code again. So, this is a way for isolation or protection also, you do not want to give access to the hardware to any program. But any hardware access, you need to go through the operating system by requesting it via a system call.

So, the system called supported by an operating system, in some sense form the application programmer interface or the API of an operating system. So, the OS exposes a bunch of functions. And user program can call these functions for various services.

So, now to avoid a user program written in one OS with one API to have to change to run on another voice. There has been some standardization. So, there is a standard API, defined called the POSIX API. So, this is for portability. That is if you have written code for one operating system, you should be able to run on another operating system also, because the API is the same. So, this POSIX API defines a standard set of system calls that have to be supported by an operating system.

And if you write a program using this API, then you can run your program on any other operating system also, that supports the same API. So, this is OS1, and this could be OS2, if it has the same API, then your same code can run on this operating system also. So, most common operating systems today are compliant with this POSIX API. So, that when you write a program on one version of Linux, it will easily run on the other version of Linux also.

So, however, you do not have to change the program. But you may have to recompile it. If the underlying architecture, if the hardware architecture is different than the CPU instructions are different, then you may have to recompile your program again. If this is x 86, and this is something else, the same instructions will not run on another hardware, you might have to recompile your program, but you do not have to rewrite it if you move to another POSIX compliant operating system.

So, that is the advantage of defining this standard API. So, anyway, if you have ever written code in any language, you would be wondering what is the system call API? I have never seen it, what could it be? So, that is because most user programs you will never directly call the system calls anyways. So, every language has a set of libraries that are provided for the convenience of users, the C programming language has the C library.

So, for example, you may have called a function like printf, to print something to screen. So, this printf function in the C library is the one that actually makes the write system call to print to screen. But you as a user, you will not directly make the system call, you will simply call printf which will take care of all the details of making system calls on your behalf. So, therefore, you as a user need not even worry about what are the system calls, what is the POSIX API. If you use your language libraries, this is automatically handled for you.

(Refer Slide Time: 14:50)

User mode and kernel mode

- Modern CPUs operate at multiple privilege levels
- User programs run in unprivileged user mode of CPU
- CPU shifts to privileged kernel mode for running OS code during:
  - Interrupts: external events
  - System calls: user request for OS services
  - Program faults: errors that need OS attention
- OS code executes in kernel mode, and returns back to user code
  - CPU switches back to low privilege level to run user code

Handwritten red text: "User code" with an arrow pointing to "unprivileged user mode".  
Handwritten red text: "kernel (OS)" with an arrow pointing to "privileged kernel mode".

So next, the next concept I would like to discuss is what is called user mode and kernel mode? We have seen that modern CPUs have multiple privilege levels, also called rings in x 86, so user programs run in an unprivileged mode. And the operating system runs in a privileged mode. So, this unprivileged mode is also called user mode. The privileged mode is also called kernel mode. So, normally your user program is running in user mode, then you will jump to kernel mode, periodically, run the OS. And then you will once again come back to user mode to run user code.

So, what causes this jump? When do you go from user mode to kernel mode and back? So, the CPU shifts to the kernel mode and runs the operating system for the following events. When an interrupt happens, when some external device requests the services of the operating system, when you have typed something on the keyboard, and that keyboard input has to be taken, the device driver has to be run.

In such cases, you will jump from the user code to the kernel code and kernel mode and run the operating system. When the user makes a system call, user himself says look, I need this work to be done. Then also you will jump into kernel mode, run the OS code and go back.

And the third case is when some error happens when the program is running some hardware says the memories hardware says look something bad has happened the user has accessed, an array out of bounds or something like that. Then also you will jump from the user core to the kernel mode run the operating system.



And once the OS has run, you have handled this event then you will go back to user mode again, the CPU will switch to a lower privileged level, you will go back to running the user code. So, these are called the user mode and the kernel mode of a CPU.

(Refer Slide Time: 16:54)

**What OS does: memory management**

- OS allocates memory for the memory image of a process
  - Memory allocated in fixed granularity of pages
  - Upon process termination, memory is freed up and assigned to other processes
- How do we assign memory addresses to process code+data?
  - How does a compiler know which memory locations will be given to process by OS?
- Process code+data are assigned virtual addresses initially
  - Compiler assigns memory addresses starting from 0 in executable
  - Later parts of memory image (stack, heap...) continue at addresses after executable
- OS maintains mapping between virtual memory addresses and real (physical) addresses in page table
  - Virtual addresses translated to physical addresses during execution using page table
- OS ensures efficient usage of memory, by allocating memory on demand

So, now we have seen process management, processes runs for some time in user mode, sometimes in kernel mode and so on. The next thing that the operating system does is memory management. So, every process that is running, we have seen that it has a memory image consisting of the code and data and the executable other things like the stack, heap, all of this is the memory image of a process in RAM. There is code there is data in RAM.

And the OS is responsible for allocating this memory for the process. And in a few lectures back we have also seen that resources like memory are usually allocated at a fixed granularity. So, most systems today allocate memory at the granularity of pages, fixed sized pages of typical sizes 4 kilobytes.

This makes it just efficient to manage the memory. So, the OS is responsible for allocating some number of memory pages to processes to store the memory image when the process is created and freeing up this memory when the process terminates. All of this memory allocation, de-allocation is handled by the operating system.

So, the next question comes up now we have seen that every piece of code and data has some address. It is located at some memory address and the CPU will request this code and data by using this address saying give me an instruction at this address load a piece of data at this

address into a register. So, the CPU refers to the memory image using addresses. But the question comes up, if we do not know the memory addresses of the OS doing the allocation, how does the user program know what address its code and data is located in?

For example, how will a compiler know when the compiler is compiling a program? How will it know at which memory location the code will be placed, at which memory location this variable will be placed? There is no way because the compiler is not doing the memory allocation, the OS is doing it.

So, how do we solve this problem? The way we solve this problem is by using what are called virtual addresses. So, initially, the compiler assumes it has a lot of memory starting from address zero and it starts putting in all the code, data, assigning, placing them in memory starting from address 0 and assigning addresses to all the code and data in this way. Then when the program starts to run the stack, heap all of these will get assigned various addresses starting from address 0.

This does not mean that they are placed at these addresses in RAM, we do not know where they will be placed in RAM. But these addresses are initially assigned both by the compiler and later on at runtime also for things like the stack and heap and these are called virtual addresses.

But in reality, this program could be placed at some other address say starting at address x in the RAM. This is the physical address, this code and data could be placed from address x to some address y in RAM. And those are the real addresses are the physical addresses which only the operating system knows, why? Because the operating system is the one allocating this memory for the program. So, the operating system knows these real addresses. And it maintains a mapping between these virtual addresses and the real addresses. And the data structure that has this mapping is called the page table.

So, when the program is running, when the CPU wants to execute some instruction or fetch some data, CPU says give me the piece of information at this memory address at that runtime, the OS ensures that these addresses are translated from these dummy virtual addresses to actual locations in RAM. So, the OS because it has all this background information of where this program is in memory, it can ensure that this translation happens correctly.

One of the main functions of an OS when it comes to memory management is also ensuring this translation of virtual addresses to physical addresses when the program is running. And

there are also other things that the OS does for example, it will allocate memory on demand it will not give a program a lot of memory because memory might run out. So, all of these things there are many other things that the operating system does with respect to memory management that we will study in this course in a lot of detail.

(Refer Slide Time: 21:32)

**What OS does: I/O management**

- OS has **device drivers** for all I/O devices connected to the system
  - Device drivers form a big part of the code of modern OS
  - Giving commands to I/O devices, interrupt handling, ...
- **File system**: OS code that takes care of storing user file data persistently on the hard disk
  - Works with hard disk device driver to store/retrieve **blocks** from disk
- **Network stack**: OS manages communication with other machines over the network

Handwritten diagrams:  
1. A diagram showing two boxes connected by a curved arrow labeled 'n/w'.  
2. A diagram showing a stack of three boxes: 'open read user' at the top, 'OS' in the middle, and 'disk' at the bottom. A box labeled 'file' is shown next to the 'disk' box.

Moving on, the next functionality of the operating system is managing I/O devices. We have seen in a previous lecture, that there are many different types of I/O devices. And each comes with a device driver that talks to the device that writes, reads from the various device registers in the device controller, issues commands, handles interrupts, sets up DMA buffers, all of this is done by the device driver.

And the OS has a bunch of device drivers for all the I/O devices connected to the system. And in addition to the device drivers, the operating system also has other pieces of code called the file system for example. So, when you store data on your hard disk, this data is stored in the form of files, all of you must be familiar with what files are.

So, then the operating system is responsible for managing these files, you as a user you will say open this file, read this file, write to this file all of this the user will keep doing. And the operating system ensures that all of this functionality is implemented on the underlying file and the piece of the OS code that does that is called the file system.

And this file system of course, works with the device driver of the hard disk to communicate with the hard disk itself and to store information on the hard disk in the form of blocks. Then

the other piece of code, the important piece of code the operating system has is the network stack that is every computer communicates with various other computers over the network either Ethernet or Wi-Fi you all must be familiar with it. So, the operating system also does a lot of this communication is also handled by the operating system in a piece of code called the network stack.

(Refer Slide Time: 23:22)

**Booting your system**

- What happens when you boot up a computer system?
- Basic Input Output System (**BIOS**) starts to run
  - Resides in non-volatile memory, sets up all other hardware
- BIOS locates the **boot loader** in the boot disk (hard disk, USB, ..)
  - Simple program whose job is to locate and load the **OS**
- Boot loader loads OS in memory and sets it up for execution
- CPU starts executing OS code
- OS exposes a terminal / shell / other interfaces to user
- User runs programs, starts processes

**Handwritten Diagram:** A flowchart on the right side of the slide. It starts with 'BIOS' at the top, with an arrow pointing down to 'boot loader'. From 'boot loader', an arrow points down to 'OS'. Below 'OS', there is a box labeled 'RAM' with '/// OS' inside it, indicating the OS is loaded into memory. A plus sign is placed between the 'OS' and the 'RAM' box.

So, one final piece of the puzzle, what happens when you boot up your system? So, when you boot up your computer system, there is a very small piece of code that runs which is the which is called the Basic Input Output System or your BIOS. This BIOS is part of your motherboard and this resides in non-volatile memory. So, that even when the power is off, the BIOS program still exists unlike RAM which is wiped clean when you turn off the power.

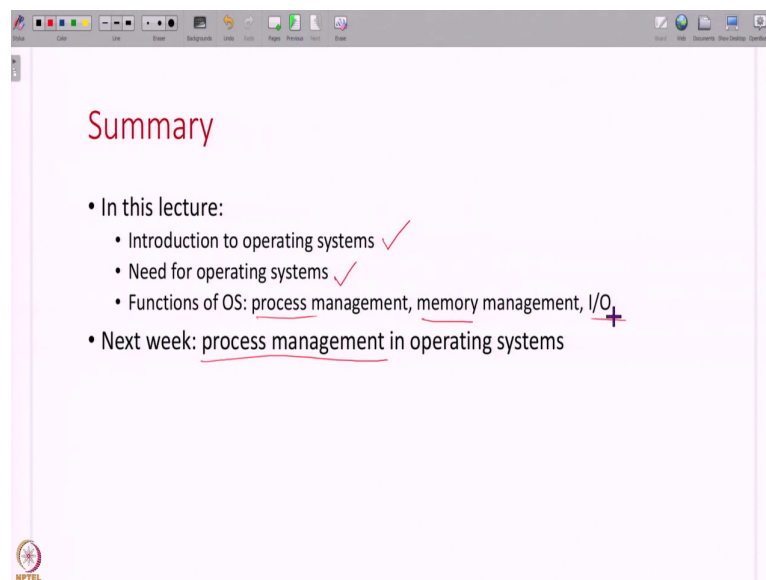
So, the BIOS program still exists when you turn on your computer. So, this program, this BIOS starts running, it sets up all the hardware, then this bios will then find the boot loader in the boot disk. So, there is one disk which is identified as the boot disk in your system, usually the hard disk or if you are booting up and was from the USB or a CD drive or DVD whatever there is a main boot disk, in that boot disk there is a program called the boot loader.

What this boot loader does is this loads the operating system. So, you have your BIOS, this BIOS starts the boot loader. It finds the boot disk which is the main disk to use and identifies and loads this boot loader. And this boot loader then searches for the full operating system and loads the operating system in memory.

So, now by the end of all of this boot process in your RAM, a part of the RAM is occupied by the operating system. And now, the RAM has the operating system and the CPU starts executing the operating system code. So, this is the end result of your boot process. When you turn on your computer to the time you see some screen or a terminal or a shell, all of this happens, the BIOS has run, the boot loader has run then the OS has been loaded and the CPU starts running the operating system code.

So, from now on, once the operating system starts, then you can run other programs, whatever your computer system is supposed to do, you start your browser, email, games, other applications, web servers, databases, whatever you want to run, they all will be run after the OS starts and all of these other processes will be managed by the operating system.

(Refer Slide Time: 25:50)



So, this is the end of this fifth lecture. In this lecture, what we have studied as we have studied what are operating systems and why do we need an operating system and then we have studied what are the various things the operating system does with respect to managing processes memory, I/O devices and so on.

And starting from next week, we are going to go deep into operating systems. We are going to begin with understanding process management in operating systems. And in the next three weeks, we are going to cover all of these topics around process, memory and I/O and operating systems.

And after that in the later part of the course once the base layer is clear, we are going to move on higher up and understand how user programs and how software systems are built in general. So, see you all next week, when we begin our study of process management and operating systems. Thank you.