SCC.131 Digital Systems Operating Systems(1)

Week(19) L2

Ibrahim Aref

2024/2025

In this lecture:

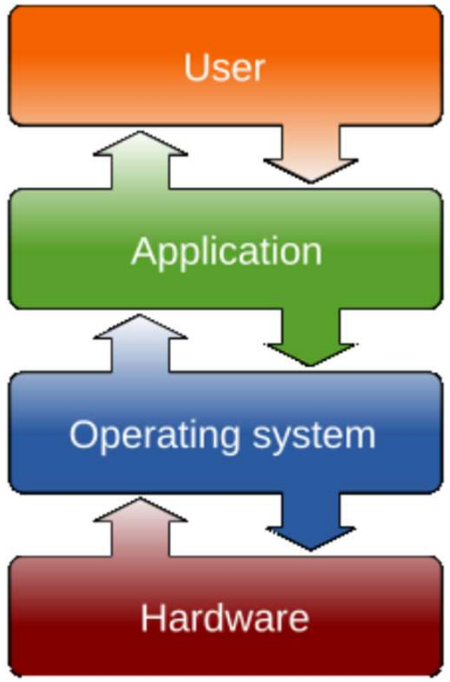
- What is an Operating System (OS)?
- Kernel - How does it work?
- Kernel-mode and User-mode.
- Processes and Threads
- Scheduling.

What is an Operating System (OS)?

- OS is a software program that manages all the hardware and software on a computer.
- Acting as the bridge between the user and the computer's physical components.
- Allowing you to interact with the device by managing tasks like file management, memory allocation, and running applications
- Coordinating all computer's resources: CPU, memory, storage devices, and peripherals (keyboard, mouse), to ensure smooth operation.
- Provides a visual interface (desktop) for users to interact with the computer and launch applications.

Important Keywords

- **Hardware:** all the physical electronic and mechanical elements forming part of a computer system. Provides basic computing resources (CPU, memory, I/O devices)
- **Software:** the instructions or programs that the hardware needs in order to function.
- **Operating system** controls and coordinates the use of the hardware among the various application programmes for various users
- **Applications** programmes define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programmes, etc.)
- **Users** (people, machines, other computers)



https://en.wikipedia.org/wiki/Operating_system

Main Tasks

- **File management:** Creating, deleting, organizing, and accessing files on the computer.
- **Memory management:** Allocating memory to different running applications.
- **Process management:** Managing the execution of multiple programs simultaneously
- **Input/Output handling:** Managing communication between the computer and external devices like printers and scanners.
- **Security & Access Control:** Protects system integrity, user data, and prevents unauthorized access.
- **Multitasking & Scheduling:** Manages task execution efficiently using scheduling algorithms.

Operating System Purpose

- Manages and allocates resources
- Control the execution of user programs and operations of I/O devices
- **Kernel**:
    - Is the core of a computer's operating system (OS).
    - It's the software that acts as an interface between the hardware and the rest of the OS.
    - Running at all times.

Examples of Operating Systems

ubuntu
FreeBSD
BlackBerry
iOS
UNIX

The kernel

Kernel

- The kernel is the core software running the system. It is “the Operating System” essentially.
- Two common design of kernels:
  - **Monolithic kernels** are single programs that contain all kernel functions. It is used in most modern OSs, including Windows, Mac OS X, and Linux.
  - **Micro-kernels** split areas of functionality up into different programs, running a minimal kernel and as much as possible running in user-space instead.
- The kernel, and your programs, run in different modes: **user-mode vs kernel-mode**.
- These terms are used to talk about the restrictions and security different types of programs can have.

How does the kernel work?

- The kernel loads into memory during startup and remains in memory until the system shuts down.
- The kernel has unlimited access to the hardware, while user applications have limited access.
- The kernel intervenes to repair errors that occur in user mode.
- Modern operating systems (Windows, Linux, and macOS) use two distinct execution modes: **Kernel Mode** and **User Mode**. These modes help ensure system security and stability by separating application-level execution from core system operations.

Kernel-Mode Code

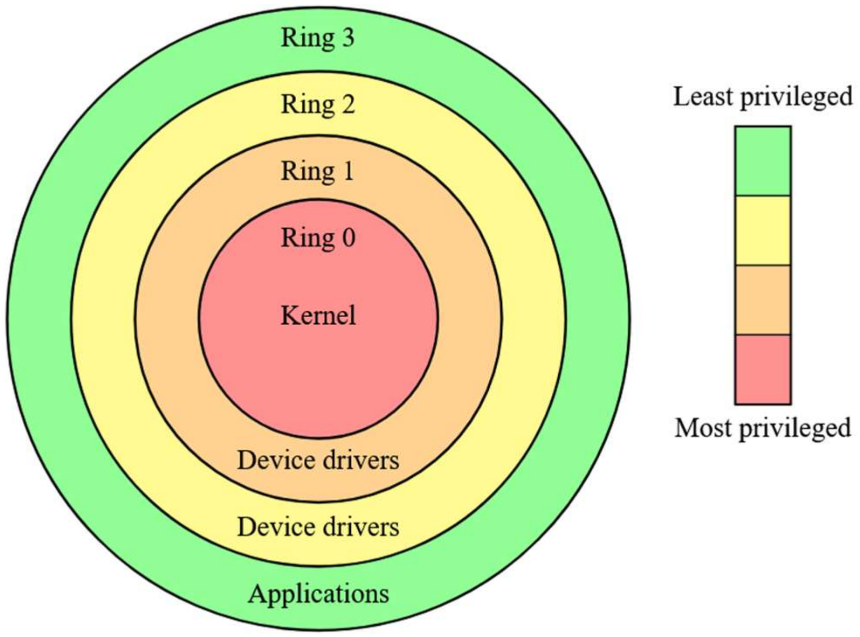
- Code running in kernel mode has unrestricted access to the machine.
    - Do anything, access anything.
    - It can directly access any memory address in the computer.
    - It can access any hardware.
- The kernel needs to run in this mode so it can control the hardware and provide the *process* abstraction for user-mode programs.
- For this to work, the hardware (CPU) needs to have a distinction between privileged and non-privileged modes of operation
- In early computers and simpler embedded systems, all the code runs in kernel mode, completely controlling the hardware with no other programs running. Downside – your program can do anything!

Protection Rings

- The **ordered protection domains** are referred to as **Protection Rings**.
- It is a security architecture used to regulate access to system resources and prevent unauthorized actions.
- Provide different privilege levels, ensuring that critical system components are safeguarded from user applications.
- This idea is implemented in different ways on different CPUs.

Structure of Protection Rings

- Most modern operating systems implement a four-ring model, though ***x86 architectures*** use only two rings (Ring 0 & Ring 3):
  - **Ring 0 (Kernel Mode):** Full access to hardware and system resources (OS kernel and device drivers).
  - **Ring 1:** Handles system services and privileged drivers.
  - **Ring 2:** Used for additional security, managing I/O operations.
  - **Ring 3 (User Mode):** Runs applications with restricted privileges (user programs and software).



Credit: Hertzsprung, Wikimedia, CC-BY-SA-3

Purpose of Protection Rings

- **Security:** Prevents unauthorized code from directly accessing hardware.
- **Stability:** Ensures user applications don’t crash the entire system.
- **Efficiency:** Controls privilege levels to manage resources effectively.
- Improving **fault tolerance**.

x86 and ARM CPU modes

- x86 CPUs implement protection rings (previous slide) as part of it’s **real/protected mode** distinction:
  - The CPU starts in *real mode* and can do anything.
  - Typically, it boots the operating system kernel program, which configures the basics of protection levels and then switches to ***protected mode****.*
  - ***Protected mode*** hardware-enforces whatever the different protection ring settings are.
  - The kernel will have a ring that allows it do to everything, there will be a ring for “normal” user programs, and there could be more levels of ring in-between depending on the operating system design (eg, a ring for **device drivers** that can do more then **user code** but less than the full kernel).
- ARM CPUs have 3/4 protection levels for user programs, kernel, hypervisor, firmware (ARM 8+).

User-mode Code

- Code running in user-mode is restricted:
  - It will only be able to access the memory the kernel has allowed it to.
  - It may not be able to call some opcodes, like changing mode!
- The concept of a user-space program running under a kernel is abstracted into the idea of a ***process***.
- In user mode, applications run with limited privileges to prevent direct access to hardware, ensuring system stability.

What happens when User-mode code attempts direct hardware access?

- An application in User-mode cannot directly access hardware. If it attempts to do so:
  - The processor will detect the unauthorized operation and generate a trap or fault (e.g., a segmentation fault).
  - The operating system’s kernel will interfere and either **terminate the offending process** or **handle the situation smoothly** (e.g., returning an error to the application).

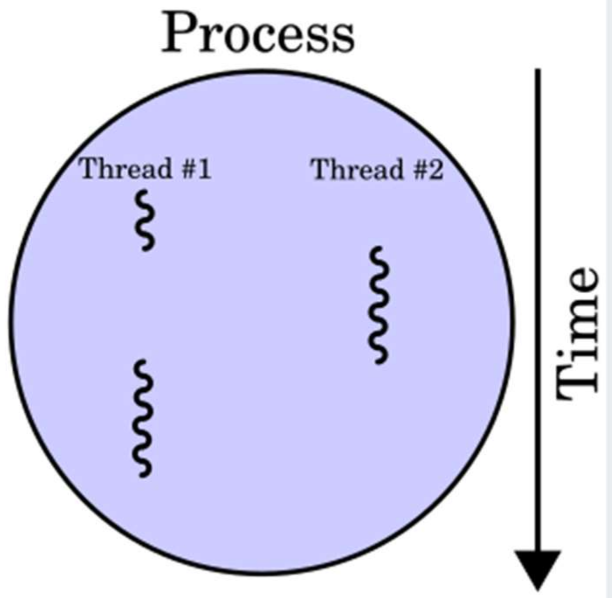
Processes

- Let’s be more specific about what we mean by when we say program.
- A *process* is a program running on the machine, managed by the OS.
- The OS will keep track of the following for a process:
    - The code it is executing
    - Memory that belongs to it, and perhaps shared access to other memory.
    - Any system resources it has - open files, network sockets, locks, etc.
    - Security permissions – what can the program access, which user is running it.
    - The hardware context

- **Process/thread/task** – caution as terminology across different OS.

Threads

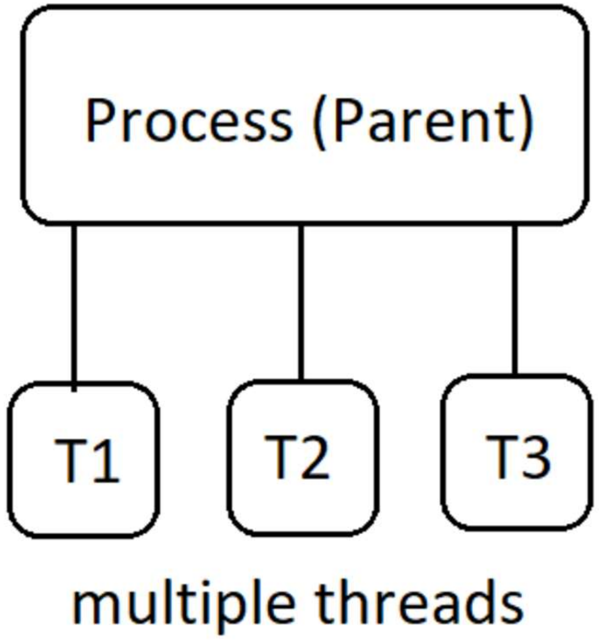
- A **process** can have **one or multiple threads**. Each thread shares the same resources as its parent process but executes independently.
  - **Lightweight**: Threads use fewer system resources compared to full processes.
  - **Shared Memory**: Threads within the same process share memory and global variables.
  - **Fast Context Switching**: Threads switch faster than processes since they don't require memory remapping.
  - **Parallel Execution**: Threads enable multitasking and parallelism within a process.



https://en.wikipedia.org/wiki/Thread_%28computing%29

Threads vs Process (1)

- The process (parent) is the main program running.
- Inside the process, multiple threads (T1, T2, T3) **run independently** but share the same memory and resources.
- This allows for **faster execution and better efficiency** compared to creating multiple processes.
- Threads improve performance by enabling **parallel execution of tasks** within a process. They are widely used in web servers, operating systems, and modern applications to make programs more efficient and responsive.



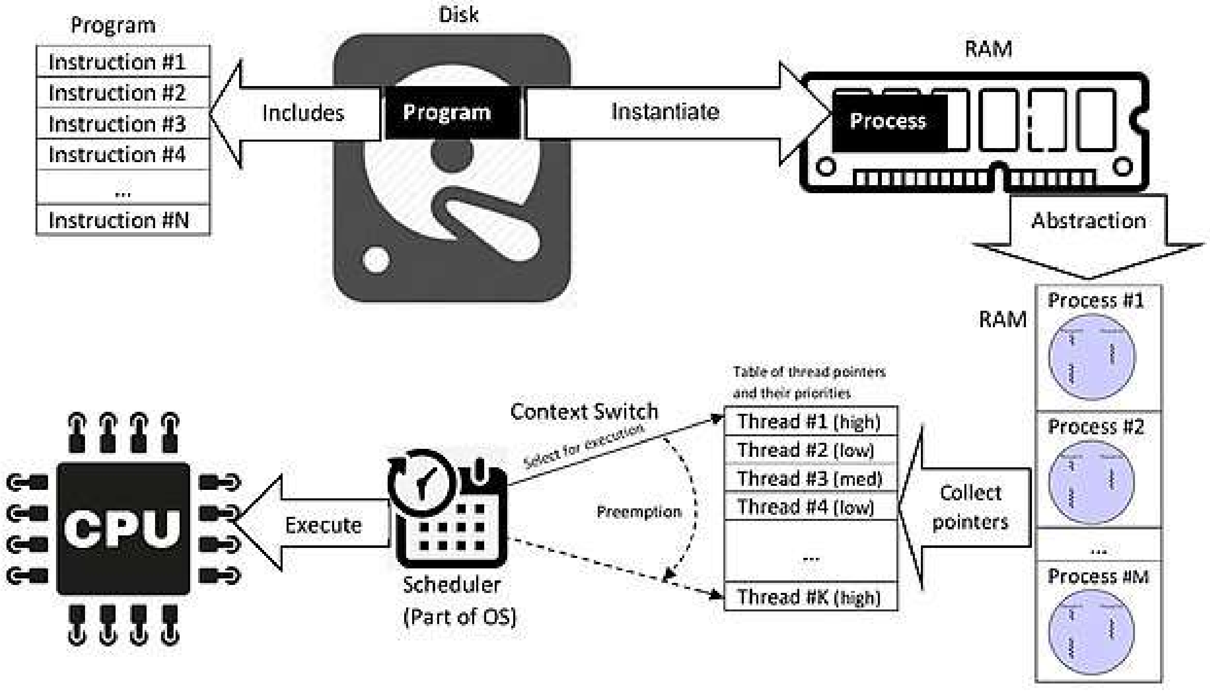
Threads vs Process (2)

Feature	Threads	Processes
Definition	is part of the process. It allows a program to perform multiple tasks simultaneously.	a program in execution
Memory Sharing	Yes, threads share process memory	No, processes have separate memory
Context Switching*	Faster	Slower
Creation Overhead**	Low	High
Example	Web brower tabs (threads)	Different applications (processes)

***Context switching** is the process of storing and restoring the state of a thread or process so that execution can resume from the same point later. It allows multiple tasks to share a CPU efficiently.

****Creation overhead** refers to the time and computational cost required to create a new thread or new process. Threads generally have lower overhead than processes because they share memory and resources, whereas processes require independent memory and resource allocation.

Program vs. Process vs. Thread



https://en.wikipedia.org/wiki/Thread_%28computing%29

Scheduling

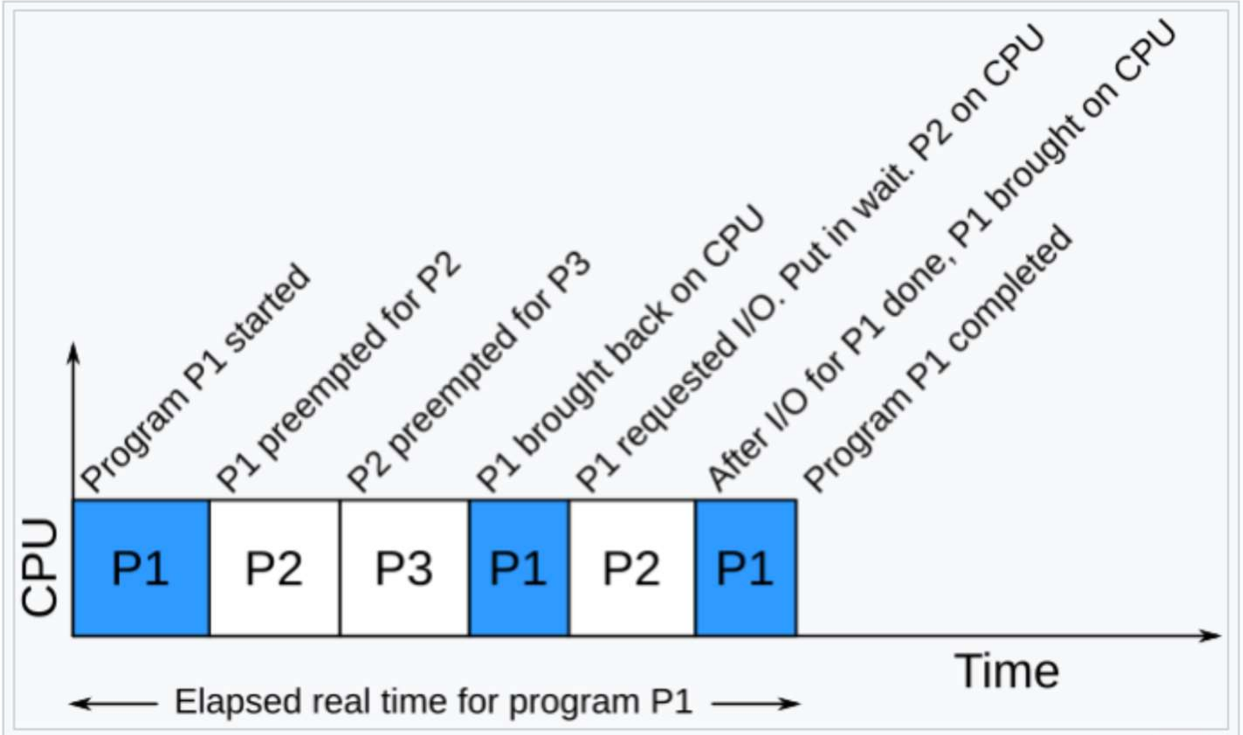
Scheduling (1)

- The method by which the OS decides **which processes or threads will be assigned to the CPU for execution**.
- It ensures **efficient CPU utilization** and **fair process execution** by managing multiple processes and threads in a multitasking environment.
- **Multitasking** allows multiple processes or tasks to run simultaneously by sharing CPU time. It **improves system efficiency** by quickly switching between tasks, making it appear as though multiple programs are running at the same time.
- **Types of Multitasking:**
  - **Pre-emptive Multitasking:** OS decides how long each task gets CPU time and forcefully switches between tasks. **E.g. Windows, Linux, macOS**.
  - **Cooperative Multitasking:** Each task voluntarily gives up control of the CPU, requiring well-behaved programs. **E.g. Older systems like Windows 3.x and classic Mac OS**.

Scheduling (2)

- For most operating systems, this means **multi-tasking** – running multiple processes on the machine interleaved over time.
  - This may be actual parallel running of processes (e.g, multiple CPUs or cores)
  - And/or **time slicing** between processes - quickly and frequently switching between which process is running to give the impression they are all running simultaneously.
  - **Time slicing can be**
    - **Cooperative** – the process indicates when it is ok to stop for a moment
    - **Pre-emptive** – the OS decides when to switch and it's done without the process's involvement.
- Most OS use **pre-emptive** multi-tasking and the part of the kernel that decides who to switch between processes is the scheduler.

Scheduling (3)



CPU Time on Single CPU Multi Tasking System

🟦 CPU color time for program P1

https://en.wikipedia.org/wiki/CPU_time

Scheduling Algorithms

- **First-Come, First-Served (FCFS)** – Non-preemptive, runs in order of arrival.
- **Shortest Job Next (SJN or SJF)** – Runs the shortest process first.
- **Round Robin (RR)** – Each process gets a fixed time slice (time quantum).
- **Priority Scheduling** – Higher priority processes execute first.
- **Multilevel Queue Scheduling** – Divides processes into different priority queues.

Types of Scheduling(1)

1- Long-Term Scheduling (Job Scheduling)

- Decides **which processes** will be admitted into the system for processing.
- Controls **degree of multiprogramming** (number of concurrent processes).
- Example: **Batch processing systems**.

2- Short-Term Scheduling (CPU Scheduling)

- Decides which process gets the **CPU next**.
- Runs frequently (every few milliseconds).
- Uses **scheduling algorithms** like **FCFS, Round Robin, SJF, Priority Scheduling**.
- Example: **Time-sharing systems**.

Types of Scheduling(2)

3- Medium-Term Scheduling

- Temporarily removes processes from memory (swapping) to **improve performance**.
- Example: **Suspending processes in memory-limited systems**.

4- I/O Scheduling

- Manages I/O requests and determines the order of **disk accesses**.
- Example: **Disk scheduling algorithms like FCFS, SSTF, SCAN**.

Disk scheduling

- Process scheduling is not the only scheduling a kernel does.
- When you try to read something from disk, the OS doesn’t necessarily do it immediately.
- There will likely be a lot of read and write requests to the disk in queue at one time. What order should they be done in?
- Needs a scheduling algorithm to know what I/O jobs to dispatch to disks in what order.
- Different devices will have different needs/algorithms.
  - **E.g.** spinning hard disk vs flash disk, since they have different read/seek properties.

Why is Scheduling Important?

- **Ensures fairness:** No process is left waiting indefinitely.
- **Maximises CPU utilisation:** Prevents CPU idleness.
- **Minimises response time:** Provides quick interaction in time-sharing systems.
- **Balances system load:** Distributes work efficiently among processors.

Interesting links

- General operating systems
  - *Modern Operating Systems*, 3rd Ed., Tanenbaum
    Online version from linbrary
    Chapter 1 covers all the operating system and virtual machine ideas we've talked about quite closely and is generally a good depth of knowledge; the whole book is often used as an Operating Systems module text (you can skip the precise details of many individual functions they talk about, section 1.7 on types of operating system goes into more depth than we do, and we're not covering filesystem that gets discussed in section 1.5.3)

Interesting links (II)

- On system calls, if you wish to explore more.
  - *Modern Operating Systems*, 3rd Ed., Tanenbaum
    System call process - Sec 1.6 p47
    System call design considerations - Sec 13.2.3 p964
  - [Linux Kernel Source]`/arch/asm/unistd.h`
    Syscall numbers for Linux
  - [Linux Kernel Source]`/arch/x85/vsdo/`
    Implementation of VSDO concept.
  - Similar discussion for Linux syscall for PPC architecture
    http://www.linuxchix.org/content/courses/kernel_hacking/lesson7
  - DragonFlyBSD
    http://dragonflybsd.org
  - Arrakis
    http://arrakis.cs.washington.edu

Summary

- Operating systems, main tasks, and the main purpose.
- Kernel and how it is working. Kernel Mode and User Mode.
- Protection Rings –why?
- Processes, Threads and Scheduling.
- **Next .....**
  - System Calls.
  - Virtual Memory.