CS2ON Operating Systems and Computer Networking

Introduction to Operating Systems and System Calls

Lecture Notes

Table of Contents

Session 1: Introduction to the Module

1.1 Module Overview

CS2ON Operating Systems and Computer Networking provides a comprehensive introduction to the fundamental concepts that underpin modern computing infrastructure. Operating systems are the essential software layer that manages hardware resources and provides services to application programs; computer networking enables communication between distributed systems. Together, these two disciplines form the backbone of virtually every computing environment, from embedded devices to cloud data centres.

This module adopts a conceptual-first approach: rather than focusing on a single operating system or networking stack, it explores the general principles and design trade-offs that apply across platforms. Students will examine how operating systems manage processes, memory, file systems, and I/O devices, and will investigate the layered architecture of computer networks. By the end of the module, students will possess the theoretical foundation needed to understand, evaluate, and work with real-world operating systems and network protocols.

1.2 Module Learning Outcomes

Upon successful completion of this module, students will be able to:

- Apply core operating system concepts — including process management, scheduling, synchronisation, and deadlock — to analyse the behaviour of real-world systems.
- Compare and evaluate CPU scheduling algorithms with respect to fairness, throughput, turnaround time, and response time.
- Understand and explain memory management techniques, including segmentation and paging, and their impact on system performance and protection.
- Compare I/O access methods and describe how operating systems interact with peripheral devices through device drivers and interrupt handling.

1.3 Module Arrangements

The module is delivered through a combination of lectures and laboratory sessions. There are approximately ten hours of lectures covering the theoretical foundations, complemented by five hours of supervised laboratory work in which students gain hands-on experience with operating system utilities, shell commands, and networking

tools. Assessment typically comprises coursework assignments and a written examination, with details confirmed on the module's virtual learning environment.

1.4 What to Expect from This Lecture

This first lecture is organised into six sessions that progressively build an understanding of operating systems from first principles. We begin with the module overview you are reading now, then explore what an operating system is and the services it provides. We examine common operating system structures — from simple monolithic designs to modern modular architectures — before discussing the critical distinction between user mode and kernel mode. We then consider the different types of users who interact with an operating system, and finally we study system calls: the mechanism by which application software requests services from the operating system kernel.

Session 2: What is an Operating System?

2.1 The Role of an Operating System

An operating system (OS) is a program that acts as an intermediary between users and the computer hardware. Its fundamental purpose is to provide an environment in which users can execute programs conveniently and efficiently. Without an operating system, every application would need to manage hardware resources directly — an impractical and error-prone proposition.

From an abstract perspective, the computing stack can be visualised as a series of layers. At the bottom sits the hardware: the central processing unit (CPU), memory, and I/O devices. Above the hardware sits the operating system, which manages these resources and presents a clean, uniform interface to the layers above. Application programs — word processors, web browsers, compilers — sit atop the OS and rely on it for resource allocation, file management, and inter-process communication. At the very top are the users, who interact with applications and, indirectly, with the operating system.

The operating system therefore serves a dual role. As a resource manager, it allocates CPU time, memory, and I/O bandwidth among competing processes fairly and efficiently. As an abstraction provider, it hides the complexities of hardware from application developers, enabling them to write portable, high-level code.

> **Key Concept — Operating System**
>
> An operating system is system software that manages computer hardware and software resources and provides common services for computer programs. It acts as an intermediary between users and hardware, offering resource management, process scheduling, memory allocation, and I/O control.

2.2 The Shell and the Kernel

An operating system comprises two principal components: the shell and the kernel. A helpful analogy is that of an office. The shell is like a receptionist at the front desk: it is the point of contact for users, accepting commands (typed or clicked) and passing them inward for processing. The kernel, by contrast, is like the office manager working behind the scenes: it handles the actual management of resources, scheduling of tasks, and communication with hardware.

The shell provides the user interface to the operating system. It may take the form of a command-line interface (CLI), where users type textual commands, or a graphical user interface (GUI), where users interact through windows, icons, and menus. In either case, the shell interprets user input, translates it into system calls, and relays the results back to the user.

The kernel is the core of the operating system. It runs in a privileged execution mode (kernel mode) with unrestricted access to hardware. Its responsibilities include process management, memory management, device driver coordination, and enforcement of security policies. Because the kernel operates at the lowest software level, its correctness and stability are paramount: a bug in the kernel can crash the entire system.

Key Concept — Shell vs Kernel

The shell is the outer layer of the OS that provides a user interface (CLI or GUI) for accepting commands. The kernel is the inner core that manages hardware resources, schedules processes, and enforces security. The shell translates user requests into system calls that the kernel executes.

2.3 Operating System Services

An operating system provides a wide range of services that simplify application development and ensure orderly, secure use of hardware resources. The following table summarises the principal services offered by a modern operating system.

Service	Description
Program Execution	The OS loads programs into memory, allocates CPU time, and manages their execution from start to termination.
I/O Operations	Applications cannot access I/O devices directly; the OS provides a uniform interface for reading from and writing to devices such as disks, printers, and network adapters.
File System Manipulation	The OS manages the creation, deletion, reading, writing, and organisation of files and directories on storage media.
Disk Management	The OS handles disk scheduling, space allocation, and defragmentation to optimise storage performance and reliability.
Memory Management	The OS tracks which parts of memory are in use, allocates and deallocates memory for processes, and implements virtual memory through paging and segmentation.

Protection and Security	The OS enforces access controls, authenticates users, and isolates processes from one another to prevent unauthorised access and ensure system integrity.
Command-Line Interface	A text-based interface (e.g., Bash, PowerShell) that allows users to interact with the OS by typing commands.
Graphical User Interface	A visual interface (e.g., Windows Desktop, GNOME, macOS Aqua) that allows users to interact through windows, icons, menus, and pointers.

Session 3: Operating System Structures

3.1 Introduction

The internal organisation of an operating system — its structure — has profound implications for its performance, reliability, maintainability, and security. Over the decades, several structural approaches have been developed, each representing a different set of trade-offs. Understanding these structures provides insight into why modern operating systems are designed the way they are and helps engineers make informed choices when building or configuring systems.

3.2 Simple Structure

The earliest operating systems, such as MS-DOS, had little or no internal structure. In MS-DOS, the interfaces and levels of functionality were not well separated: application programs could access the basic I/O routines directly, and there was no clear distinction between user-mode and kernel-mode code. This lack of separation meant that a misbehaving application could easily crash the entire system.

Simple-structure operating systems were designed for a single user on limited hardware. They offered minimal protection, no multitasking, and no memory isolation. While adequate for the personal computers of the early 1980s, this approach is wholly unsuitable for modern multi-user, networked environments.

3.3 Monolithic Structure

A monolithic kernel places all operating system services — process management, memory management, file systems, device drivers, and networking — into a single, large block of code that runs entirely in kernel mode. UNIX, Linux, and traditional Windows NT kernels are examples of monolithic (or predominantly monolithic) designs.

The principal advantage of a monolithic kernel is performance: because all services reside in the same address space, communication between them involves simple function calls rather than expensive inter-process communication (IPC). The disadvantage is complexity and fragility: a bug in any part of the kernel — say, a faulty device driver — can bring down the entire system.

> **Key Concept — Monolithic Kernel**
>
> A monolithic kernel is an operating system architecture in which all system services run in a single address space in kernel mode. This approach maximises performance through direct function calls but increases the risk that a single bug

can crash the entire system.

3.4 Layered Structure

The layered approach organises the operating system into a hierarchy of layers, each built on top of the one below. The lowest layer (layer 0) is the hardware; the highest layer is the user interface. Each layer uses only the services provided by the layer immediately below it, and provides services to the layer above.

The key advantage of layering is modularity and ease of debugging: each layer can be developed and tested independently. If a layer functions correctly using only the services of the layer below, then errors are localised to the layer being debugged. However, defining the layers appropriately is challenging, and the strict layering can introduce performance overhead due to the need to traverse multiple layers for common operations.

3.5 Microkernel Structure

A microkernel keeps only the most essential services — such as inter-process communication (IPC), basic scheduling, and low-level address space management — in the kernel. All other services, including file systems, device drivers, and networking, run as user-space processes. Examples include the Mach microkernel (which forms the basis of macOS) and MINIX.

The microkernel approach offers significant benefits for reliability and security: if a device driver crashes, it does so in user space and can be restarted without affecting the kernel. The system is also more extensible, as new services can be added without modifying the kernel. The trade-off is performance: communication between services requires message passing through the kernel (IPC), which is slower than the direct function calls used in monolithic kernels.

3.6 Modular Structure

Most modern operating systems, including Linux and Solaris, adopt a modular approach that combines the best features of monolithic and microkernel designs. The core kernel provides essential services and defines interfaces through which additional functionality can be loaded dynamically as kernel modules. For instance, a device driver or file system can be compiled as a loadable module and inserted into a running kernel without requiring a reboot.

The modular approach preserves the performance advantages of a monolithic kernel (modules run in kernel space and communicate via direct calls) while providing the flexibility and maintainability benefits of a microkernel (functionality can be added, removed, or updated independently).

3.7 Comparison of OS Structures

The following table compares the five structural approaches discussed above.

Structure	Example(s)	Advantages	Disadvantages
Simple	MS-DOS	Easy to implement on limited hardware; minimal overhead.	No protection or isolation; a single fault crashes the system.
Monolithic	UNIX, Linux, Windows NT	High performance due to direct function calls within a single address space.	Complex codebase; a bug in any component can crash the entire system.
Layered	THE (Dijkstra), OS/2	Clear separation of concerns; each layer can be tested independently.	Difficult to define layers appropriately; performance overhead from traversing multiple layers.
Microkernel	Mach, MINIX, QNX	High reliability and security; services can be restarted independently.	Performance overhead from IPC message passing between user-space services.
Modular	Modern Linux, Solaris	Combines monolithic performance with modular flexibility; supports dynamic loading.	Modules run in kernel space, so a faulty module can still crash the system.

Session 4: User Mode and Kernel Mode

4.1 Why Two Execution Modes?

Modern operating systems distinguish between at least two execution modes to protect the integrity and stability of the system. If all code — both application programs and the operating system itself — ran with the same privileges, a bug or malicious action in any program could corrupt kernel data structures, overwrite memory belonging to other processes, or communicate with hardware in uncontrolled ways. Dual-mode operation prevents this by restricting the operations that application code can perform.

4.2 User Mode

When a process executes in user mode, it operates within a restricted environment. The CPU enforces limitations: user-mode code cannot execute privileged instructions (such as those that directly access hardware or modify page tables), and it can only access its own allocated memory. If a user-mode process attempts a privileged operation, the CPU raises an exception, and the operating system typically terminates the offending process.

The benefit of this restriction is isolation: if a user-mode application crashes — due to a null pointer dereference, an infinite loop, or a segmentation fault — only that application is affected. The operating system and all other processes continue to run normally. This isolation is fundamental to the stability of multi-user, multi-tasking systems.

4.3 Kernel Mode

When executing in kernel mode (also called supervisor mode or privileged mode), code has unrestricted access to all hardware and memory. The operating system kernel runs in this mode, enabling it to manage processes, configure memory protection, handle interrupts, and communicate directly with I/O devices.

Because kernel-mode code has no restrictions, bugs in the kernel are far more dangerous than bugs in user-mode applications. A null pointer dereference in the kernel, for example, can cause a kernel panic (on Linux/macOS) or a Blue Screen of Death (on Windows), bringing down the entire machine.

> **Key Concept — Dual-Mode Operation**
>
> Dual-mode operation divides CPU execution into user mode (restricted privileges, safe for applications) and kernel mode (full hardware access, reserved for the OS). A hardware mode bit in the CPU tracks the current mode. This separation protects

the operating system from errant or malicious application code.

4.4 Mode Switching

The transition from user mode to kernel mode is triggered by a system call, an interrupt, or an exception. When a user-mode process invokes a system call (for example, to read a file), the CPU switches to kernel mode, executes the requested service within the kernel, and then switches back to user mode before returning control to the calling process.

The mechanism relies on a hardware mode bit in the CPU. When the bit is set to 0, the CPU is in kernel mode; when set to 1, it is in user mode. The system call instruction (e.g., SYSCALL on x86-64 or SVC on ARM) atomically switches the mode bit and transfers control to a predefined kernel entry point. Upon completion of the kernel service, the return instruction restores the mode bit to user mode.

4.5 Crash Implications by Mode

The table below contrasts the impact of a crash in user mode versus kernel mode.

Aspect	User Mode	Kernel Mode
Privilege Level	Restricted: cannot execute privileged instructions.	Unrestricted: full access to all hardware and memory.
Crash Impact	Only the offending process is terminated; the OS and other processes are unaffected.	The entire system may crash (kernel panic / BSOD); all processes are lost.
Example Scenario	A web browser encounters a segmentation fault and closes.	A faulty device driver dereferences a null pointer, causing a kernel panic.
Recovery	The user can simply restart the application.	A full system reboot is typically required.

Session 5: Types of System Users

Not all users interact with an operating system in the same way. Understanding the different categories of users helps clarify the diverse requirements that an operating system must satisfy and the different levels of abstraction at which people work with computing systems.

5.1 End Users

End users interact with the computer through application programs and typically have no knowledge of — or need for — the underlying operating system mechanisms. They use word processors, web browsers, email clients, and other applications to accomplish their tasks. The operating system is invisible to them: it provides the platform on which applications run, but end users neither configure it nor write code that interacts with it directly.

From the end user's perspective, a good operating system is one that makes applications responsive, reliable, and easy to use. End users are the largest category of users, and their needs — simplicity, stability, and security — drive many operating system design decisions.

5.2 Application Programmers

Application programmers (or software developers) write the programs that end users interact with. They do not modify the operating system itself, but they rely heavily on the system APIs (Application Programming Interfaces) that the OS provides. Through these APIs, application programmers can open files, create processes, allocate memory, send network packets, and perform countless other operations without needing to understand the hardware-level details.

Application programmers value a rich, well-documented, and stable API. The POSIX standard, for example, defines a portable operating system interface that enables application code to be compiled and run on any POSIX-compliant system (e.g., Linux, macOS, various UNIX variants) with minimal modification.

5.3 System Programmers

System programmers work at the lowest level of the software stack. They develop the operating system kernel, device drivers, compilers, system utilities, and other infrastructure software. System programmers require a deep understanding of hardware architecture, memory management, interrupt handling, and concurrency. They write code that runs in kernel mode and must ensure that it is correct, efficient, and secure,

because errors at this level can compromise the stability of the entire system.

System programmers also develop the tools and libraries that application programmers depend on. Their work forms the foundation on which all higher-level software is built.

5.4 Comparison of User Types

User Type	Interaction Level	Example Activities
End User	Application level: interacts through GUIs and application features.	Writing documents, browsing the web, sending emails, playing media.
Application Programmer	API level: uses system calls and libraries provided by the OS.	Developing desktop apps, mobile apps, web services, games.
System Programmer	Hardware/kernel level: writes or modifies OS components and drivers.	Developing kernel modules, writing device drivers, building compilers.

Session 6: System Calls

6.1 What is a System Call?

A system call is the programmatic interface through which a user-space process requests a service from the operating system kernel. Because user-mode code cannot directly access hardware or kernel data structures, system calls provide a controlled, secure gateway for operations such as reading a file, creating a process, or allocating memory.

System calls are fundamental to the operation of any modern operating system. Every time a program opens a file, prints to the screen, or communicates over a network, it is — directly or indirectly — invoking one or more system calls.

> **Key Concept — System Call**
>
> A system call is a request made by a user-space process to the operating system kernel for a privileged operation. It is the primary mechanism by which applications interact with the OS, crossing the boundary from user mode to kernel mode in a controlled manner.

6.2 How a System Call Works

When a program invokes a system call, a well-defined sequence of events occurs. Consider the common scenario of a C program calling printf(), which ultimately needs to write characters to the screen:

- **Step 1:** The application calls a library function (e.g., printf() in the C standard library).
- **Step 2:** The library function prepares the arguments and invokes the appropriate system call wrapper (e.g., write()).
- **Step 3:** The wrapper places the system call number and arguments into designated CPU registers.
- **Step 4:** A special trap instruction (e.g., SYSCALL on x86-64) is executed, switching the CPU from user mode to kernel mode.
- **Step 5:** The kernel's system call handler looks up the system call number in a dispatch table and invokes the corresponding kernel function.
- **Step 6:** The kernel function performs the requested operation (e.g., writing bytes to the display device).
- **Step 7:** The kernel places the return value in a register and executes a return-

from-trap instruction, switching back to user mode.

- **Step 8:** Control returns to the library function, which returns the result to the application.

6.3 Categories of System Calls

System calls can be grouped into six broad categories:

- **File operations** — create, open, read, write, close, and delete files; manage file attributes and permissions.
- **Process control** — create, terminate, and wait for processes; load and execute programs.
- **Device manipulation** — request and release devices; read from and write to devices.
- **Information maintenance** — get and set the current time or date; get and set system data.
- **Communication** — create and delete communication connections; send and receive messages.
- **Protection** — set and get file permissions; control access to resources.

6.4 System Call Implementation

Most programming languages provide a system call interface — typically implemented in C or C++ — that allows application programs to invoke system calls without needing to know the low-level details. Each system call is assigned a unique number, and the operating system maintains a system call table that maps these numbers to the corresponding kernel functions.

When a system call is invoked, the caller need not understand how the kernel implements the service. The caller simply needs to know the interface: what parameters to pass and what return value to expect. This abstraction decouples the API from the implementation, allowing the kernel to change its internal workings without breaking application code.

6.5 Example: File Copy System Call Sequence

To illustrate how system calls work in practice, consider a simple program that copies the contents of one file to another:

- **open()** — Open the source file for reading. The OS verifies that the file exists, checks permissions, and returns a file descriptor.
- **open()** / **create()** — Open (or create) the destination file for writing.
- **read()** — Read a block of data from the source file into a memory buffer.

- **write()** — Write the buffer contents to the destination file.
- **Loop** — Repeat the read/write cycle until the entire source file has been copied.
- **close()** — Close both file descriptors, releasing the associated kernel resources.

6.6 Windows vs UNIX System Calls

The following table compares system call categories across Windows and UNIX platforms.

Category	UNIX / Linux	Windows
Process Control	fork(), exec(), exit(), wait(), kill()	CreateProcess(), ExitProcess(), WaitForSingleObject(), TerminateProcess()
File Manipulation	open(), read(), write(), close(), unlink()	CreateFile(), ReadFile(), WriteFile(), CloseHandle(), DeleteFile()
Device Manipulation	ioctl(), read(), write()	DeviceIoControl(), ReadFile(), WriteFile()
Information Maintenance	getpid(), alarm(), sleep(), time()	GetCurrentProcessId(), SetTimer(), Sleep(), GetSystemTime()
Communication	pipe(), shmget(), mmap(), socket(), send(), recv()	CreatePipe(), CreateFileMapping(), MapViewOfFile(), socket(), send(), recv()
Protection	chmod(), chown(), umask()	SetFileSecurity(), SetSecurityDescriptorOwner()

Despite the differences in naming conventions and specific semantics, the underlying concepts are the same: both platforms provide mechanisms for process control, file and device manipulation, information retrieval, inter-process communication, and access protection.

6.7 Key Takeaways

This session has introduced three fundamental ideas that underpin the interaction between application software and the operating system.

> **Key Concept — The OS as Bridge**
>
> The operating system serves as a bridge between user applications and hardware. It provides a stable, uniform interface (the system call API) that shields applications from the complexities and variations of the underlying hardware.

Key Concept — Kernel Responsibilities

The kernel is responsible for all critical low-level operations: process scheduling, memory management, device driver coordination, and security enforcement. It runs in a privileged mode with unrestricted hardware access, making its correctness essential to system stability.

Key Concept — System Calls as Safe Gateway

System calls are the controlled mechanism by which user-space processes request kernel services. They enforce the boundary between user mode and kernel mode, ensuring that applications cannot bypass the operating system’s protection mechanisms.

References and Further Reading

- GeeksforGeeks (2025) Operating Systems. Available at: https://www.geeksforgeeks.org/operating-systems/ (Accessed: February 2026).
- Silberschatz, A., Galvin, P.B. and Gagne, G. (2013) Operating System Concepts. 9th edn. Hoboken, NJ: Wiley.
- Tanenbaum, A.S. and Bos, H. (2014) Modern Operating Systems. 4th edn. Upper Saddle River, NJ: Pearson.