Part 1: Operating System Services and Interfaces

The Operating System (OS) is essentially a helper that sits between you and the computer hardware. It provides services to make the computer easier for you to use, and it manages the system in the background to keep it running efficiently.

1. Services for the User

The OS provides several visible functions to help you and your programs:

- **User Interface:** It gives you a way to interact with the computer.
- **Program Execution:** It takes a program, loads it into the computer's memory, and runs it.
- **I/O Operations:** It manages inputs and outputs, allowing your programs to use devices like keyboards, mice, screens, or USB drives.
- **File-System Manipulation:** It allows you and your programs to read, write, create, search, and delete files and folders.
- **Communications & Error Detection:** It allows different programs to share information with each other, and it constantly watches for hardware or software errors to take corrective action and prevent crashes.

2. Services for the System (Efficiency)

To keep the computer running smoothly, especially when multiple things are happening at once, the OS does a lot of background work:

- **Resource Allocation:** When multiple apps are open, the OS fairly distributes CPU time, main memory, and file storage to each of them.
- **Accounting:** It keeps track of exactly how much of the computer's resources are being used by different users.
- **Protection and Security:** It controls access to system resources so programs don't interfere with each other, and it protects the system from unauthorized outside access.

3. Types of User Interfaces

How do you actually give commands to the OS? It is usually done in one of three ways:

- **GUI (Graphical User Interface):** This is what you are likely using right now. It uses a desktop metaphor with a mouse, keyboard, monitor, icons, and folders.
- **CLI (Command-Line Interface):** A text-based interface where you directly type specific commands (often into a "shell") to tell the computer what to do.
- **Batch:** A method where the computer executes a series of programs automatically, one after the other, without needing any human interaction.

CLI and GUI are the broad categories or styles of interfaces. The actual software programs you run to experience those interfaces are the specific implementations: a "shell" is a specific program that gives you a CLI, and a desktop environment (like Mac's "Aqua" or the standard Windows desktop) is a specific program that gives you a GUI.

Part 2: System Calls & APIs

Programs are not allowed to access computer hardware directly. When a program needs something (like opening a file or sending a message), it must ask the Operating System to do it. This is done using System Calls and APIs.

1. System Calls System calls are the specific programming interface to the services provided by the OS. They are necessary because they safely switch the computer from "user mode" to "kernel mode", allowing the OS to execute the requested action. System calls are grouped into five main categories: Process control, File management, Device management, Information maintenance, and Communications.

2. APIs (Application Programming Interfaces) System calls can be complicated and differ from system to system. Because of this, programmers rarely use them directly,. Instead, they use an API (like Win32 for Windows, POSIX for UNIX/Mac, or the Java API).

Here is why APIs are used instead of direct system calls:

- **Portability:** An API remains stable, making it easier to move your software to different computer platforms.
- **Simplicity:** APIs are easier to use and provide more useful functionality than raw system calls,.
- **Stability:** The OS can update its system calls without breaking the programmer's code, because the API handles the updates,.

How it works behind the scenes: The OS maintains a table where each system call has a specific number.

- **The API (The Messenger):** Your program uses an API (which is managed by a run-time support library) to simply ask for a service. The API does not do the actual work.
- **The System Call (The Secure Bridge):** It is not just a passive interface; making a system call acts as a special instruction that safely triggers the system to switch from user mode to kernel mode. The system-call interface then looks up the requested command in a table and passes the request securely inward.
- **The OS Kernel (The Worker):** The OS kernel is the part that *actually executes* the requested task. Once the kernel finishes the work, the system-call interface returns the status and any results back to your program.

Any action you take in a **GUI** or **CLI** eventually results in a system call. In fact, many of these interfaces are simply system programs designed specifically to help users make system calls without needing to know how to code.

Your comparison is spot on:

- An **API** is a translator designed for a *program* to easily make system calls.
- A **CLI** or **GUI** is a translator designed for a *human* to easily make system calls.

When you double-click a file in a GUI, or type a command to open a file in a CLI, that interface interprets your human action and fires off the exact same "open file" system call to the OS kernel behind the scenes.

3. Passing Parameters When a program makes a system call, it must give the OS specific data (parameters), like the name of the file it wants to open. This is done in three ways:

1. **Registers:** Passing parameters directly in the CPU registers. This is the simplest method, but registers have limited space.
2. **Block/Table:** The parameters are stored in a block of memory, and the OS is only given the memory address. This is used by Linux and Solaris.
3. **Stack:** The program "pushes" parameters onto a stack, and the OS "pops" them off to read them. *Note: Both the block and stack methods are great because they do not limit the amount of data you can pass.*

Part 3: OS Structures

This is all about how the massive amount of code that makes up an Operating System is actually organized.

OS Design Goals

Before building an OS, designers must balance two sets of goals:

User Goals: The OS should be convenient, easy to learn, reliable, safe, and fast.

System Goals: For the developers, the OS should be easy to design, implement, maintain, flexible, and efficient.

When designers build an OS, they start with an important rule: separate the **Policy** (deciding *what* needs to be done) from the **Mechanism** (deciding *how* to do it). This keeps the system flexible if they need to make changes later.

Over the years, Operating Systems have been built using five main structural designs:

1. **Simple Structure:** Everything is mixed together in one place without being divided into modules. MS-DOS was designed to squeeze the most functionality into the least amount of space. However, because its interfaces and functionality were not well separated, programs could easily bypass the OS and access the hardware directly.

2. **The Original UNIX Structure:** The original UNIX had very limited structuring and basically consisted of just two parts: user-level system programs and a massive kernel. This kernel handled a huge number of functions (CPU scheduling, memory management, file systems) all tightly packed into one level, which made it quite messy.
3. **Layered Approach:** The OS is organized into strict levels, like an onion. The bottom layer (Layer 0) is the hardware, and the highest layer is the user interface. The strict rule of this "onion" design is that each layer can use functions and services *only* from the layer directly below it, while providing services to the layer above it. While this is very organized, it is extremely rigid.
4. **Microkernel:** This design keeps the core OS (the kernel) as tiny as possible by moving most services out into the "user space". It is very secure and reliable, but can be slightly slower because the different parts have to communicate by passing messages back and forth which is slower than making direct system calls.
5. **Modular Structure:** This is what most modern systems (like Solaris) use. It uses an object-oriented approach where each core component is separate. It is similar to the layered approach, but much more flexible because modules can communicate with each other freely over known interfaces, and they are loaded dynamically into the kernel only when they are actually needed.

Part 4: Virtual Machines (VMs)

A Virtual Machine takes the layered approach to the extreme by treating software as if it were actual, physical hardware. It creates the exact illusion that a guest process has its own processor and virtual memory. A well-known specific example is the **Java Virtual Machine (JVM)**, which uses a class loader and an interpreter to let Java programs run on any host system (Windows, Linux, etc.).

How it works (Host vs. Guest): The main physical computer runs a **Host Operating System**. A virtualization layer sits on top of this, allowing you to install multiple **Guest Operating Systems** (like running Windows and Linux simultaneously on the same machine). Each guest is provided with a virtual copy of the computer.

Simple Visual: [Applications] -> [Guest OS)] -> [Virtualization Layer] -> [Host OS] -> [Hardware]

History and Benefits: VMs first appeared commercially in IBM mainframes in 1972. Today, they are incredibly popular for several reasons:

- **Consolidation:** Instead of running many underused physical computers, you can unite multiple systems onto fewer, more powerful machines to save resources.
- **Protection:** The VMs are safely isolated from each other. However, you can still permit controlled file sharing or allow them to communicate with each other via networking.
- **Testing & Development:** They provide a perfect, risk-free environment for building and testing software.
- **Portability:** Using the "Open Virtual Machine Format", you can easily move a VM between completely different platforms.

Part 5: Debugging and Booting

1. Debugging (Finding and Fixing Errors) When things go wrong, the OS helps developers figure out why by saving "snapshots" of what was happening at the exact moment of the error.

- **Core Dump:** If a normal *application* crashes, the OS saves a file containing the memory that the specific app was using at the time.
- **Crash Dump:** If the *Operating System itself* crashes, it saves a file containing the kernel's memory. (They are entirely different than the checkpoints in note 4)

Think of a checkpoint as saving your progress in a video game, whereas a crash dump is the error report generated when the game suddenly freezes and closes.

2. Booting (Starting the Computer) When your computer is turned off, the OS is just resting on your hard drive. Booting is the process of getting the OS kernel loaded into memory so it can actually run.

- **The Bootstrap Program:** This is a tiny piece of startup code permanently stored right on the computer's hardware (ROM/firmware).
- **How it works:** When you press the power button, the bootstrap program wakes up. Its only job is to locate the OS kernel on the hard drive, load it into the main memory, and start it.
- *Note:* Sometimes this uses a two-step process, where a tiny "boot block" on the disk helps load the main bootstrap loader first.

Extra Info:

Here are the remaining specific details and examples from the slides that we did not cover in our main overview:

OS Services & Interfaces

- Communication between processes can happen in two specific ways: via shared memory or through message passing.
- The Graphical User Interface (GUI) was originally invented at the Xerox PARC company.

System Calls & APIs

- System calls are typically written in C or C++.
- Behind the scenes, the run-time support library handles the system calls. For example, using the standard C library, calling the `printf()` function in a program automatically triggers the `write()` system call.
- There are direct equivalents for system calls between Windows and Unix. For example, to create a process, Windows uses `CreateProcess()` while Unix uses `fork()`; to open a file, Windows uses `CreateFile()` and Unix uses `open()`.

System Programs

- System programs are divided into specific categories: file manipulation, status information (like a configuration registry), file modification (text editors), programming-language support (compilers and debuggers), program loading and execution (loaders), and communications.
- Most users' entire view of the operating system is defined by these system programs, not the actual underlying system calls.

OS Structures

- The original UNIX system consisted of just two parts: system programs and a massive kernel that handled everything below the system-call interface and above the physical hardware.
- Mac OS X uses a specific structure containing an application environment on top of a kernel environment (made up of BSD and Mach).
- The Solaris modular approach uses a core kernel surrounded by loadable modules like file systems, executable formats, and device drivers.

Virtual Machines

- The Java Virtual Machine (JVM) specifically uses a "class loader" and a "Java interpreter" to execute `.class` program files on any host system.
- VMware is an example of an architecture that allows a host OS (like Linux) to run multiple guest OSes (like FreeBSD, Windows NT, and Windows XP) simultaneously.

Debugging & Generation

- **Kernighan's Law:** The slides quote, "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it".
- **DTrace:** A tool used in Solaris, FreeBSD, and Mac OS X that allows live instrumentation on production systems by firing "probes" to capture real-time state data.
- **SYSGEN:** Before an OS boots, a program called SYSGEN is used to obtain information about the specific hardware configuration so the OS can adapt to that specific computer site.