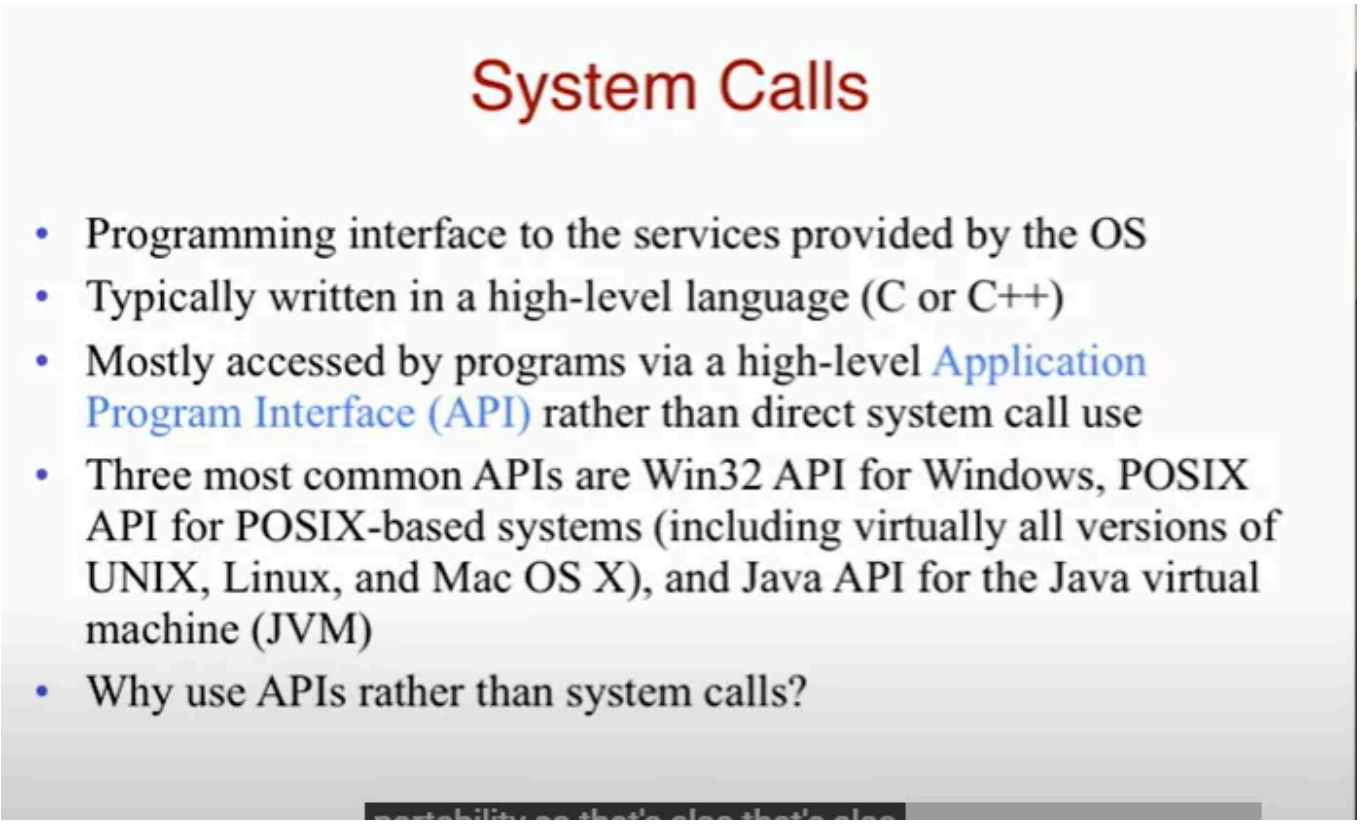
Lecture 3

OS Structures & Services

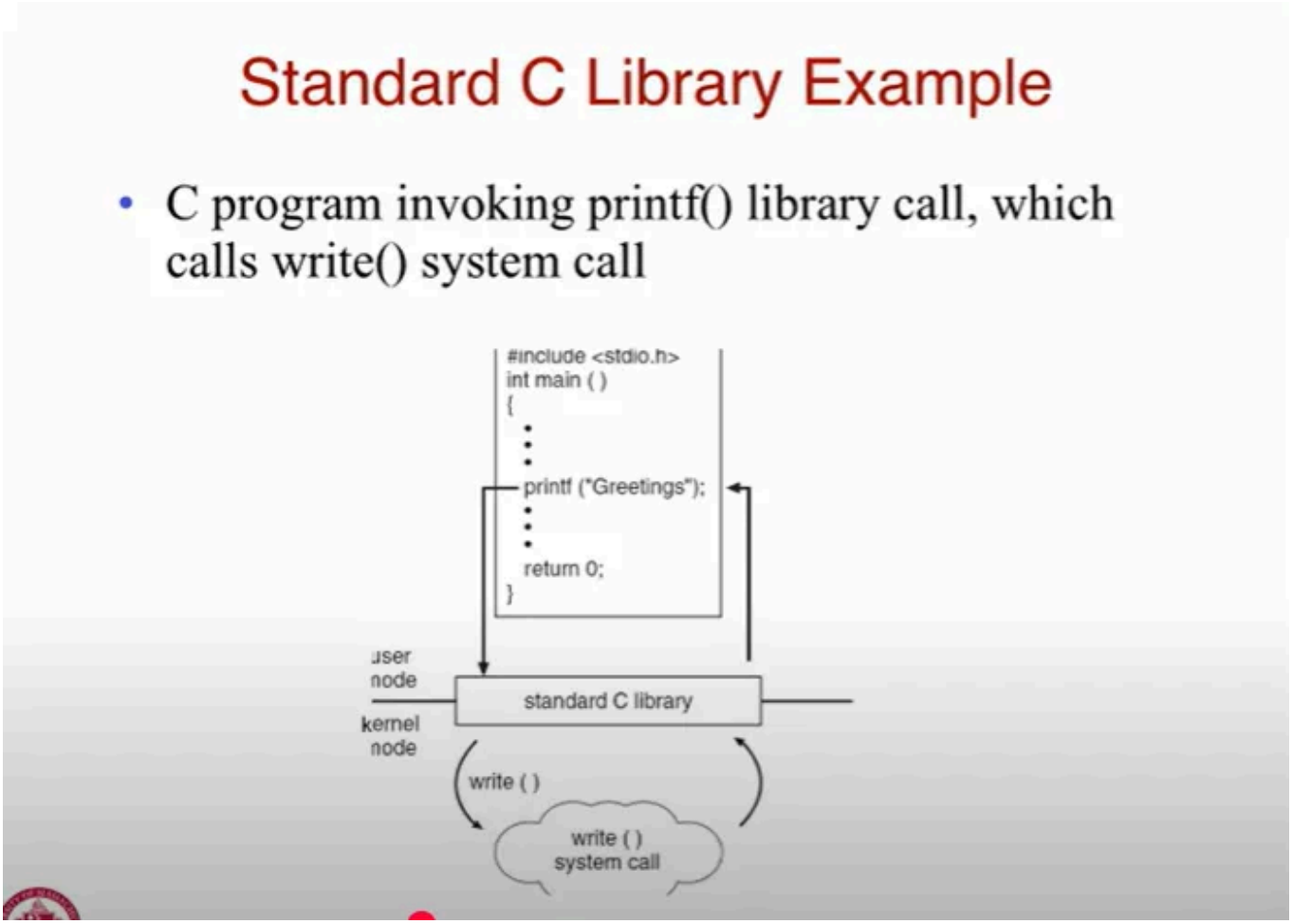
A 64 bit OS means it can work with 64 bit operations and works with a 64 bit processor.
Advantages: it can perform high degree scientific calculation and 32 bits addressing lead to only 4gb RAM and with 64 bit we can 16 exabytes(16 BIILLION GB) of RAM.
we have a co-processor along with the main processor which can offload the strain from the main CPU making the CPU more powerful and efficient. E.g. graphics/video card:-offload rendering of videos from CPU.

System Calls



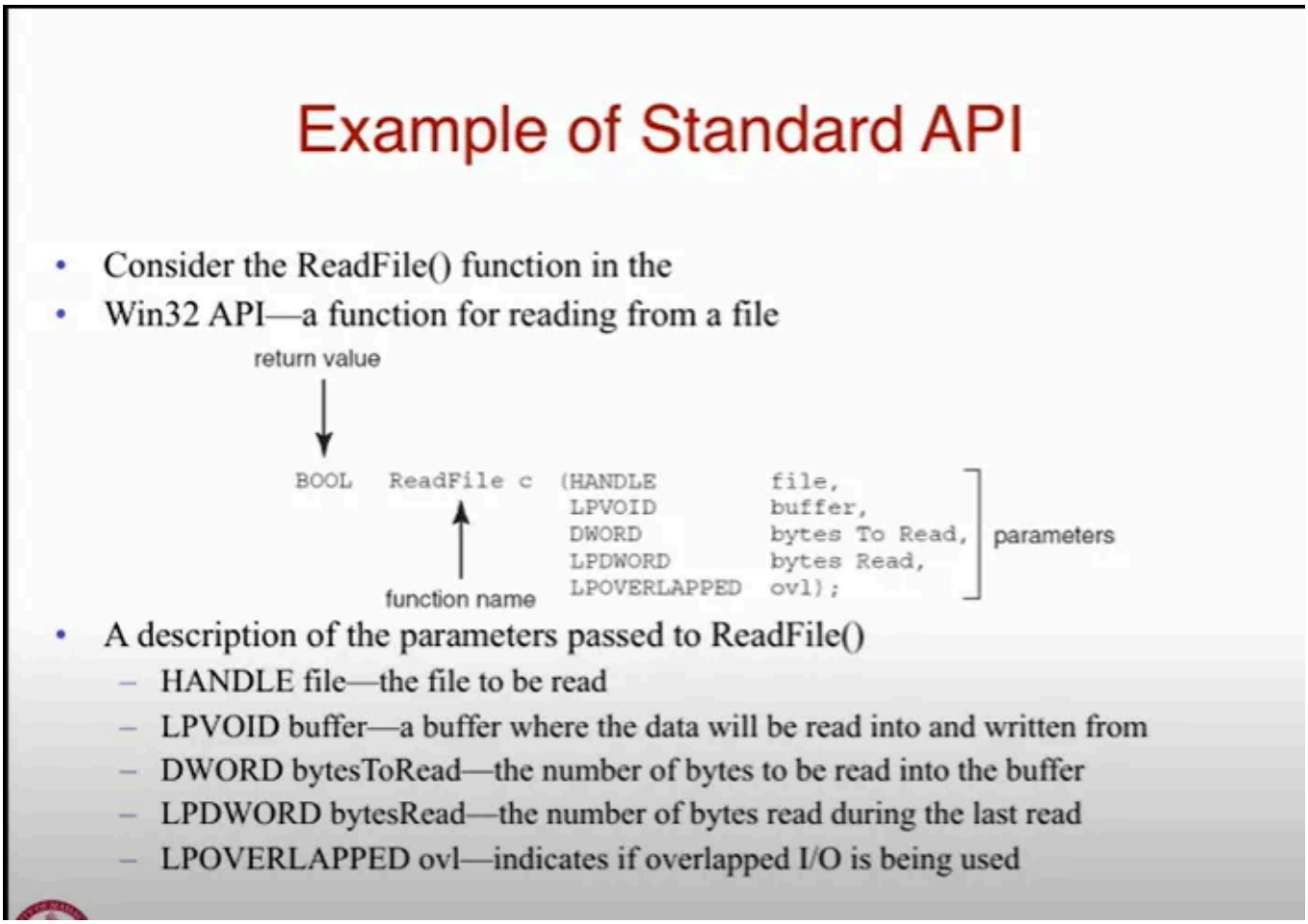
- System calls are issued when user need to perform a low hardware level task then we issue a system call to the kernel.
- System calls can be written in C and CPP but these system calls vary from OS to OS and system to system but these API provide abstraction and provide machine to machine portability without worrying about the low level hardware details.
- Typically we don't program in system calls directly, we don't write a program and actually issue a system call directly to the OS, instead, we use an API. These APIs essentially wraps all the system call we are making in an interface that's simpler and easier to use.

    - For windows: Win32 API
    - For POSIX-based systems(Linux and Mac OS): POSIX API
    - Java API for JVM
- APIs act as an intermediary between two applications, allowing them to send and receive data, or interact and perform tasks. Means we can ask for services and data from it without actually knowing how the services were provided E.g. weather app API. here API provide functions.
- APIs hide the complexity of the underlying system (like hardware or OS functions) from developers.
- APIs are portable meaning it allows applications to run on different systems without modifications.
- APIs are modularity meaning it allows the OS and applications to be developed and updates independently.
- APIs act as a gatekeeper, ensuring that applications only access resources they are allowed to use. APIs provide a consistent way for applications to request services from the OS.

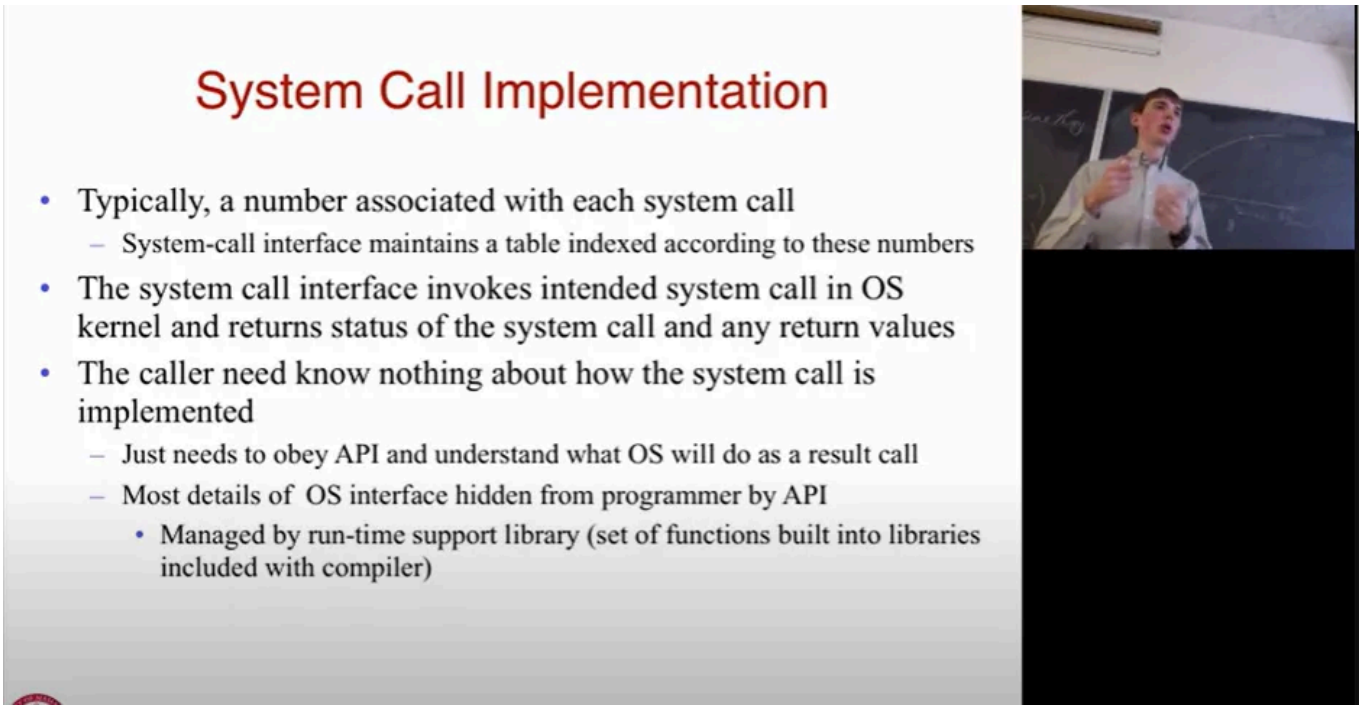


we have written a c code to print on std console. In the stdio lib the printf function is defined and in its source code. A write system call is issued which displays things on our computer. when

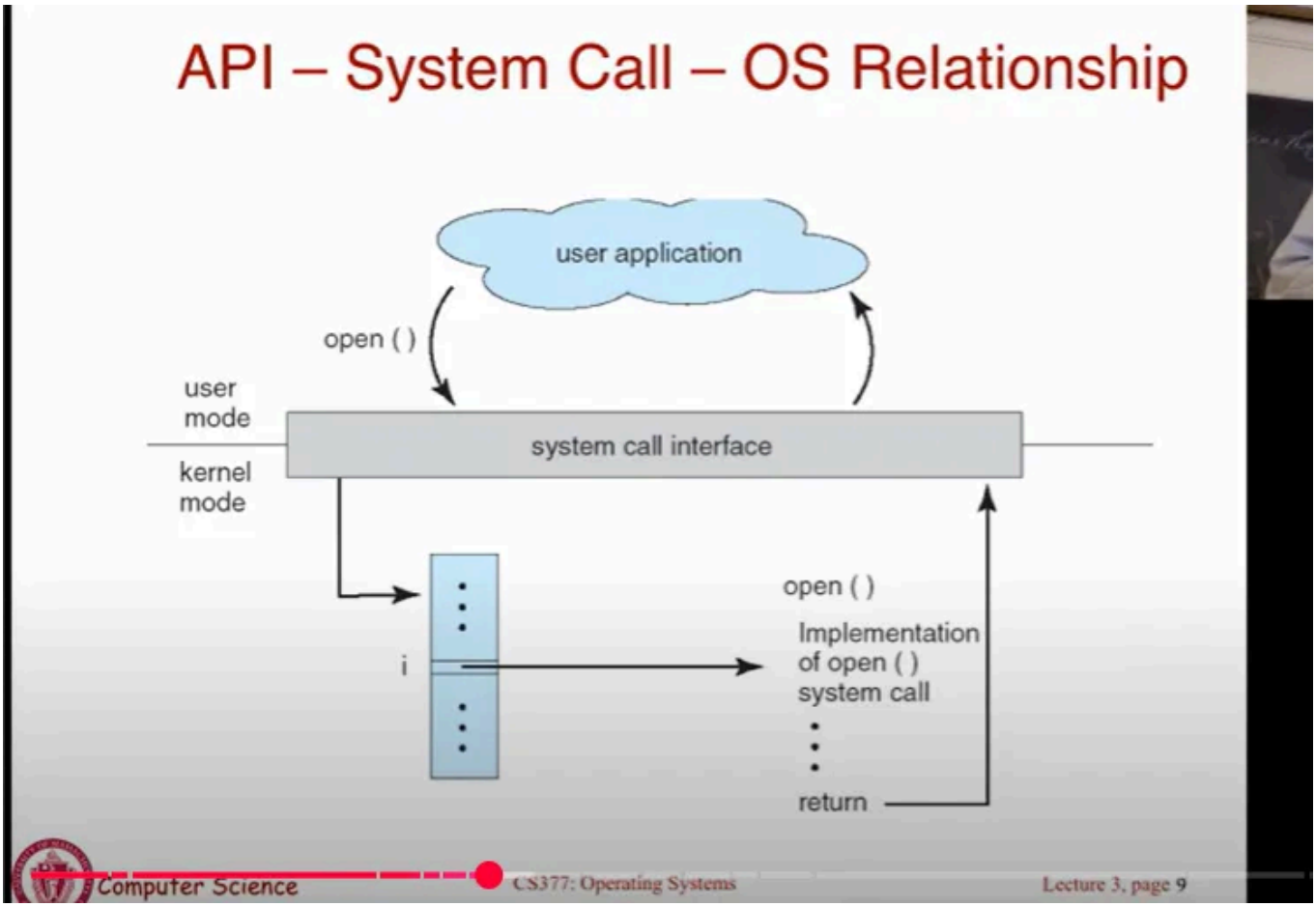
the printf call is issued (then a trap is raised which makes a jump from user to kernel mode and kernel mode takes the system call) and then process writes hello world on console and come back to user mode.



here we have a read file functions with parameters and we don't have to worry about the underlying system calls in the function.



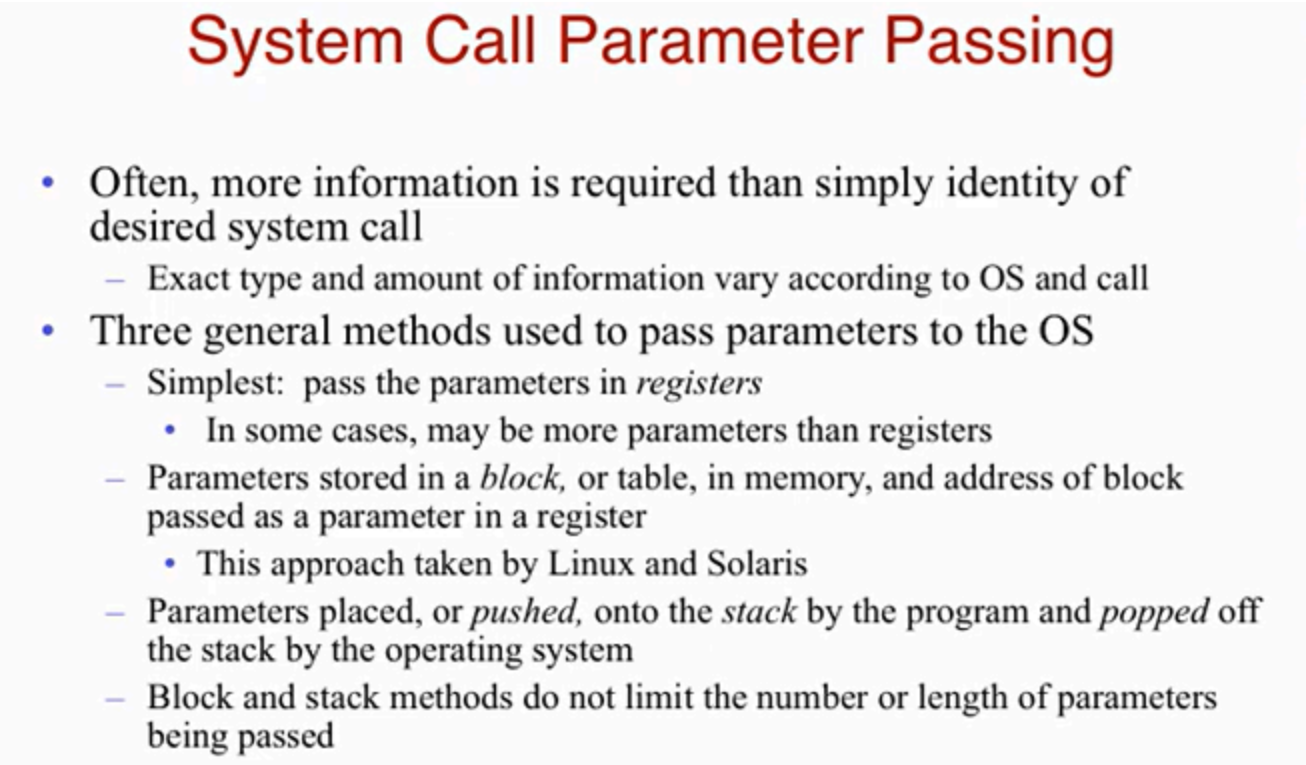
Generally every system call has a identifying number which tells what type of system call it is. when system call is invoked, we are basically passing in the number which tells what type of system call we want. The syscall interface matches the sys call and matches the code which performs the task in kernel mode and return the result. The kernel uses this number to look up the corresponding system call handler in a **dispatch table**.
just we don't give shit about how things are happening!!!



But for invoking the system call we just can't pass the identifying number through the kernel as the OS needs to maintain the Security of one program with respect to another while threading . The kernel validates whether the calling process/thread has the necessary permissions for the requested operation (e.g., accessing a file, allocating memory). Without this validation, a rogue process could exploit raw system call numbers to gain unauthorized access to system resources.

The system call interface in modern operating systems is carefully designed to balance performance, security, and isolation. The system call number identifies the operation, but the kernel ensures security and correctness by validating arguments, maintaining thread/process isolation, and enforcing privilege levels. This design prevents one thread or program from

accidentally or maliciously affecting another, even in complex multi-threaded environments.



Need for Parameter Passing: When a program makes a system call, it often needs to provide more information than just the identity of the desired system call. This additional information can include data like file names, buffer sizes, or operation modes.

Methods of Passing Parameters:

a. **Passing Parameters in Registers**:

- The simplest method is to pass the parameters directly in the CPU registers. Registers are fast storage locations within the processor.
- However, this method has limitations because there are a finite number of registers, and some system calls may require more parameters than available registers.

b. **Using a Block or Table in Memory**:

- When there are more parameters than registers, the parameters can be stored in a block or table in memory.
- The address of this block is then passed in a register. This approach is used by operating systems like Linux and Solaris.
- This method allows for passing a larger number of parameters without being constrained by the number of registers.

c. **Using the Stack**:

- Parameters can be placed (pushed) onto the stack by the program. The OS then pops these parameters off the stack when the system call is executed.
- The stack is a Last-In-First-Out (LIFO) data structure, making it suitable for this purpose.

- Like the block method, using the stack does not limit the number or length of parameters that can be passed.

Virtual Address Space

A virtual address space is an abstraction of physical memory provided by the operating system (OS). Instead of processes directly accessing physical memory (RAM), they interact with a virtual memory model, where each process gets its own isolated address space. process are given a virtual memory space and when they need the memory they are mapped to the physical memory it helps and simplifying system design isolation and main advantage is if the ram become full then it bring it can swap out things which are not currently in use

Example: Virtual Memory in Action

Scenario:

1. You are running a text editor and a web browser on your computer.
2. Both processes request memory:
   - Text editor: Needs memory to load a document.
   - Web browser: Needs memory for tabs and rendering pages.

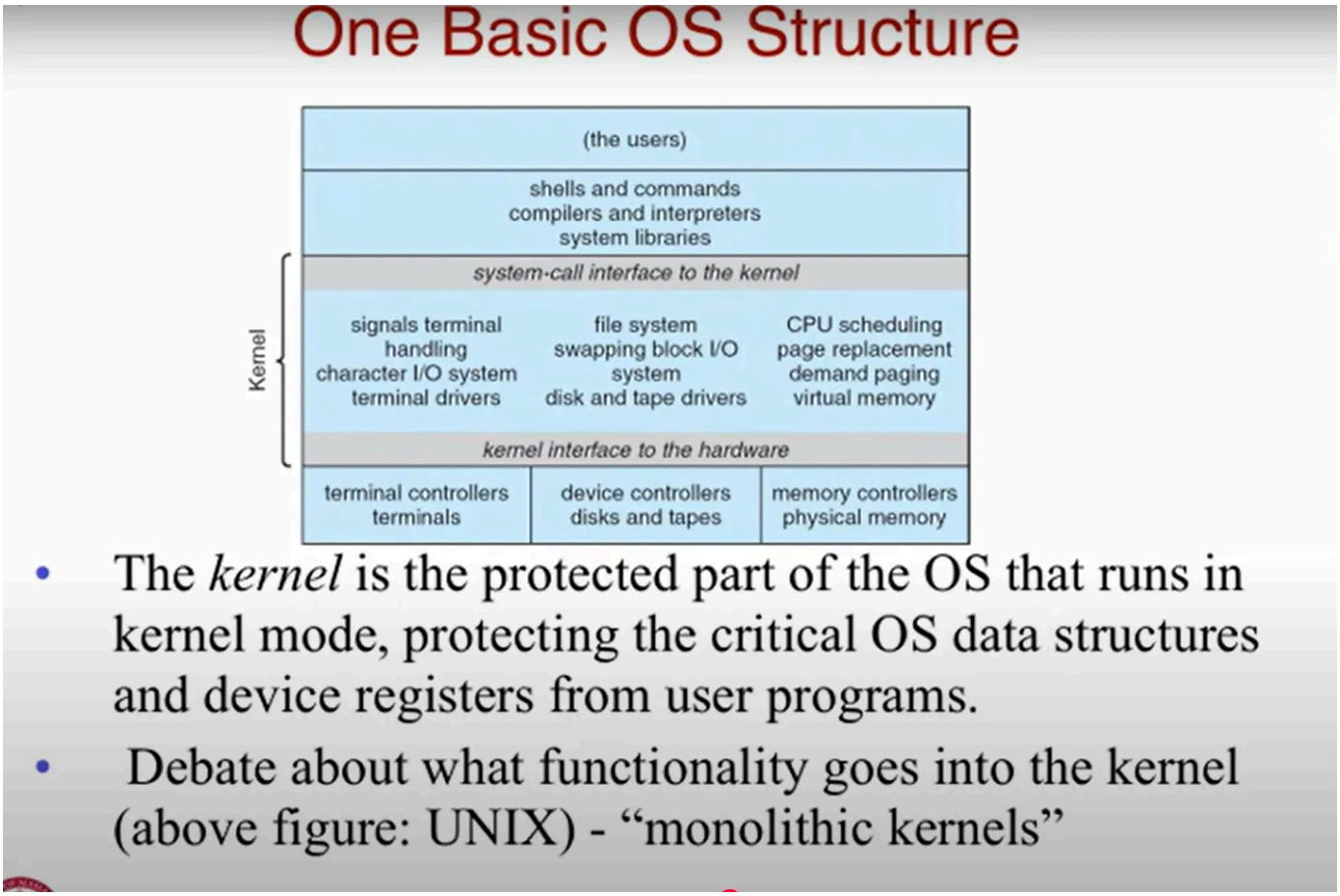
How Virtual Memory Handles It:

1. Virtual Address Space Assignment:
   - Text editor gets a virtual address space: 0x0000 to 0xFFFF.
   - Web browser gets a separate virtual address space: 0x0000 to 0xFFFF.
   - Each process sees the same range of addresses, but they map to different physical locations in RAM.
2. Address Translation:
   - When the text editor accesses virtual address 0x1000, the MMU translates it to physical address 0xABC0.
   - When the web browser accesses virtual address 0x1000, the MMU translates it to physical address 0xDEF0.
3. Page Swapping (if needed):
   - If RAM is full, the OS may move unused pages (e.g., from the web browser) to disk and update the page table.
   - When the web browser needs its data again, a page fault occurs, and the OS retrieves it from the disk.
   every process is always allocated a virtual address space. Which then is converted to physical space.

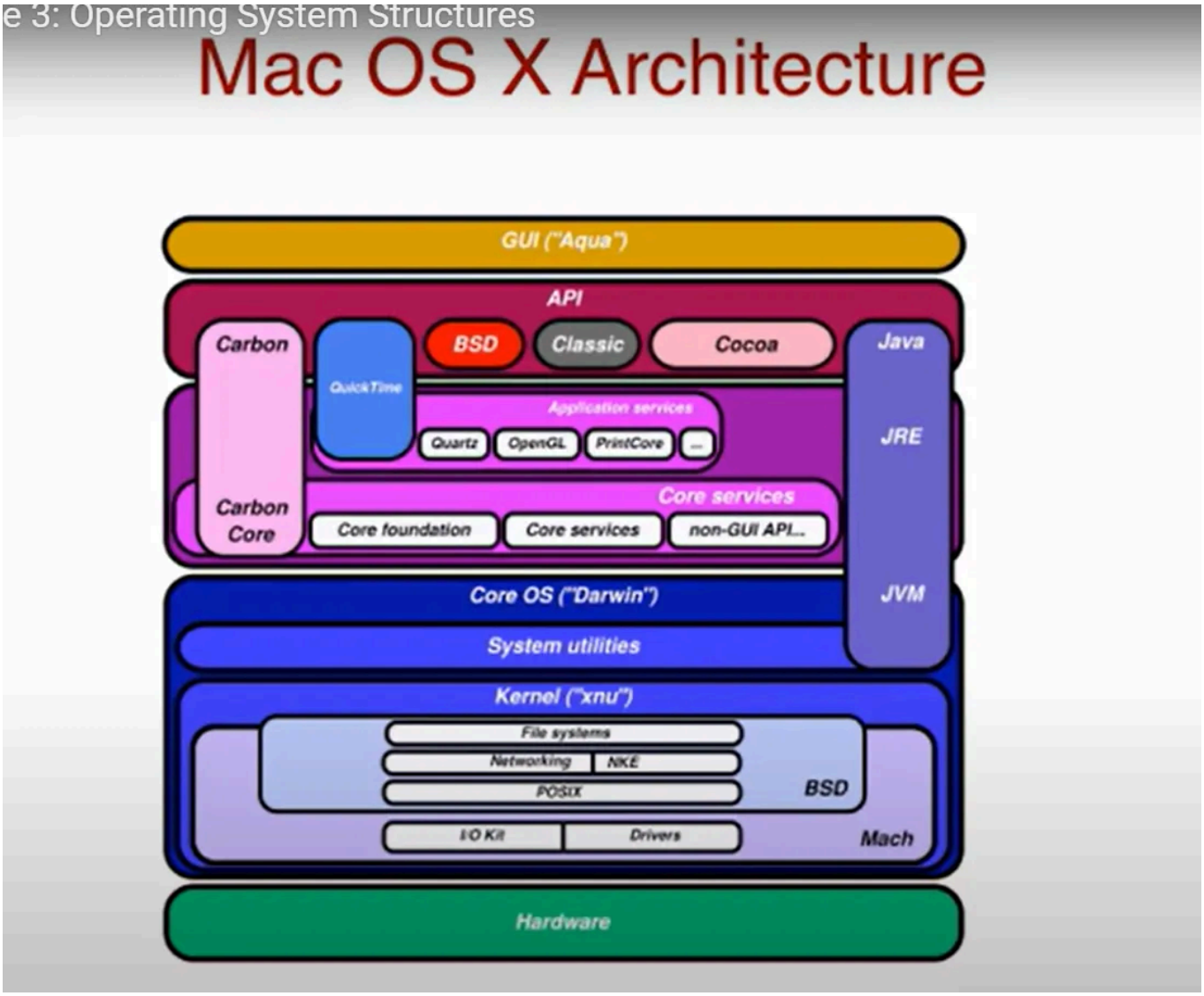
And hence arises challenges while directly passing parameters in system calls:

- User programs and the kernel run in separate address spaces for security and stability. This separation prevents user programs from directly interfering with the kernel or other processes.
- Challenge of Passing Parameters:
  - When a user program passes a parameter to the kernel (like a pointer to memory), it's in the user's virtual address space.
  - The kernel uses a different address space, so a memory address from the user program may not directly map to a valid location in the kernel's space.

OS Structure



Monolithic Kernels:
A monolithic kernel is an operating system architecture where the entire OS, including core functions and device drivers, operates in a single address space in kernel mode. This tight integration allows for efficient communication and high performance, as there's minimal overhead. However, it also increases complexity and maintenance challenges. Bugs in one part can affect the entire system, posing reliability and security risks. Examples include Linux and traditional UNIX systems. While monolithic kernels are fast and efficient, their design can make them harder to debug and secure compared to alternatives like microkernels.



The image illustrates the **Mac OS X Architecture**, showing the layers and components that form the operating system. Below is a detailed explanation of each layer:

---

1. Hardware Layer (Bottom Layer)

- Represents the physical hardware components of the system, such as the CPU, memory, storage devices, and peripherals.
- All higher layers interact with the hardware through drivers and abstractions provided by the kernel.

---

2. Kernel Layer (`xnu`)

- The core of the operating system, responsible for managing hardware resources, system calls, and low-level services.
- **Components:**
  - **Mach:** Provides low-level system services like memory management, scheduling, and interprocess communication (IPC).
  - **BSD (Berkeley Software Distribution):** Provides POSIX APIs, file systems, networking, and user permissions.
  - **I/O Kit:** A framework for managing hardware devices and drivers in an object-oriented way.
  - **Drivers:** Device drivers that directly control hardware components.

---

3. Core OS Layer (`Darwin`)

- The open-source foundation of macOS, combining the Mach kernel, BSD components, and additional system utilities.
- **Key Features:**
  - Provides fundamental system services such as file systems, networking, and process management.
  - Implements security features and provides system utilities for applications.

---

4. Core Services Layer

- Contains libraries and frameworks that provide essential system functionality for applications.
- **Components:**
  - **Core Foundation:** A low-level API providing basic data structures and utilities.
  - **Core Services:** Higher-level non-GUI APIs that support networking, file access, and other system-level services.

---

5. Application Services Layer

- Provides APIs and frameworks for graphics, printing, and application development.
- **Components:**
  - **Quartz:** A 2D graphics rendering engine for GUI elements.

- **OpenGL:** A 3D graphics rendering framework.
- **PrintCore:** Manages printing services and APIs.
- **QuickTime:** A framework for multimedia processing (audio, video, and animation).

---

6. API Layer

- The application programming interface (API) layer provides tools and frameworks for application developers.
- **Key Components:**
  - **Carbon:** A legacy framework for transitioning from older Mac OS systems to Mac OS X.
  - **Cocoa:** The modern framework for creating macOS applications, providing GUI and other services.
  - **BSD and Classic:** Provides compatibility with UNIX and legacy Mac applications.
  - **Java:** Includes Java Runtime Environment (JRE) and Java Virtual Machine (JVM) for running Java applications.

---

7. GUI Layer (`Aqua`)

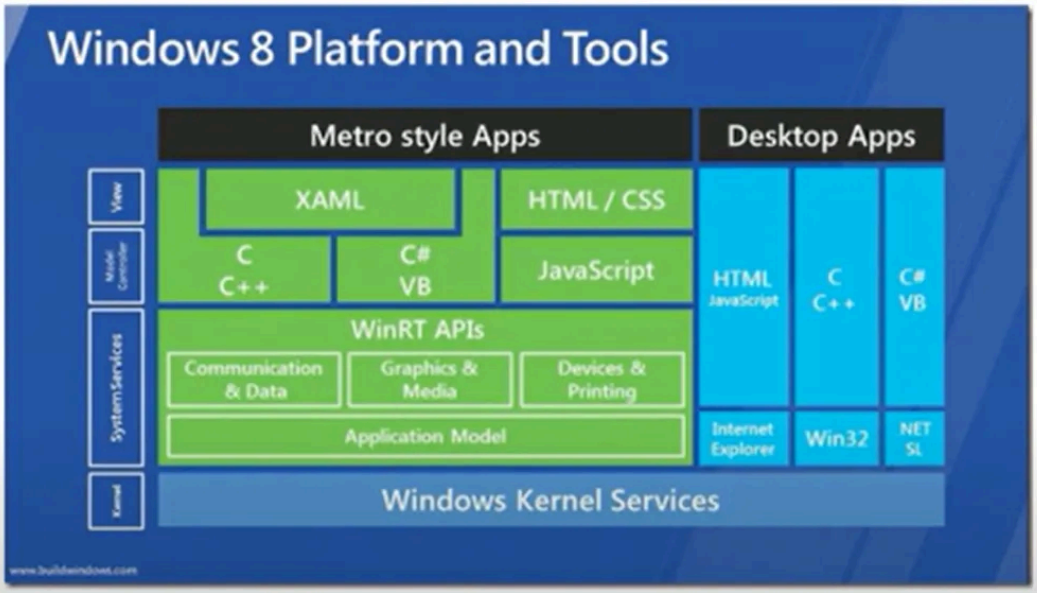
- The topmost layer, responsible for the graphical user interface (GUI) that users interact with.
- **Aqua:** The visual design and user interface of macOS, known for its distinctive aesthetic, animations, and ease of use.

Key Takeaways:

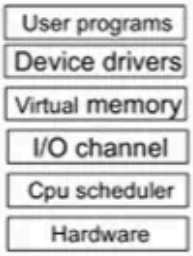
- The architecture is modular, with the **kernel** and **Darwin** forming the foundation.
- **Cocoa** and **Aqua** are integral to the macOS user experience.
- The design emphasizes compatibility (with UNIX, Java, and legacy applications), multimedia capabilities, and graphical performance.

This layered structure ensures robustness, security, and flexibility while maintaining a user-friendly experience.

Windows 8 Architecture



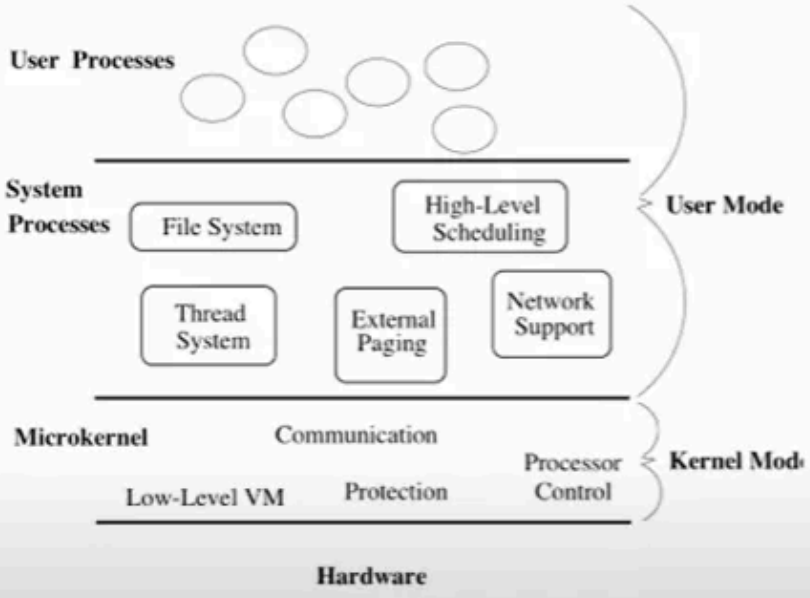
Layered OS design



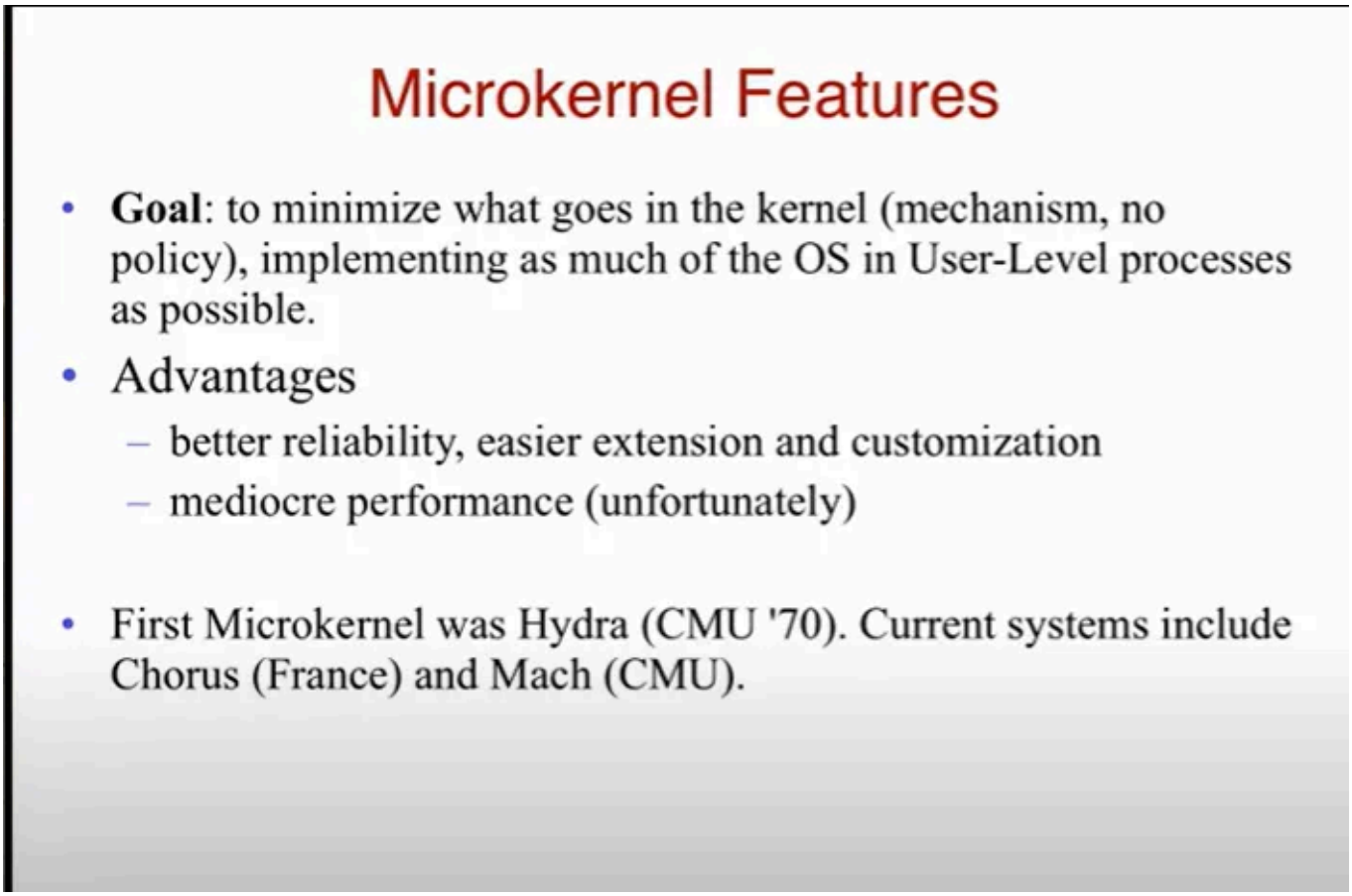
Layer N: uses layer N-1 and provides new functionality to N+1

- Advantages: modularity, simplicity, portability, ease of design/debugging
- Disadvantages communication overhead between layers, extra copying, book-keeping, layer design

Microkernel



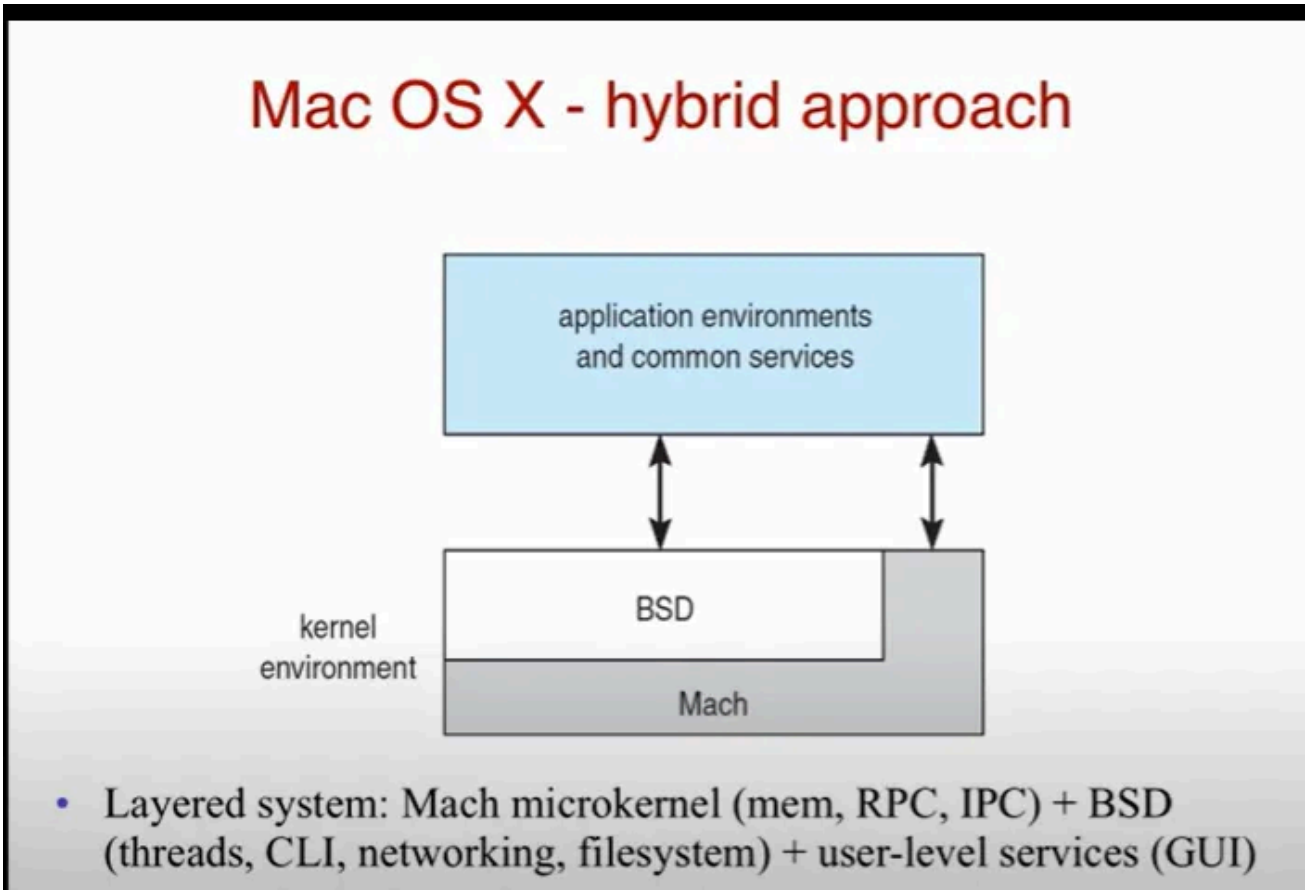
- Small kernel that provides communication (message passing) and other basic functionality
  - other OS functionality implemented as user-space processes



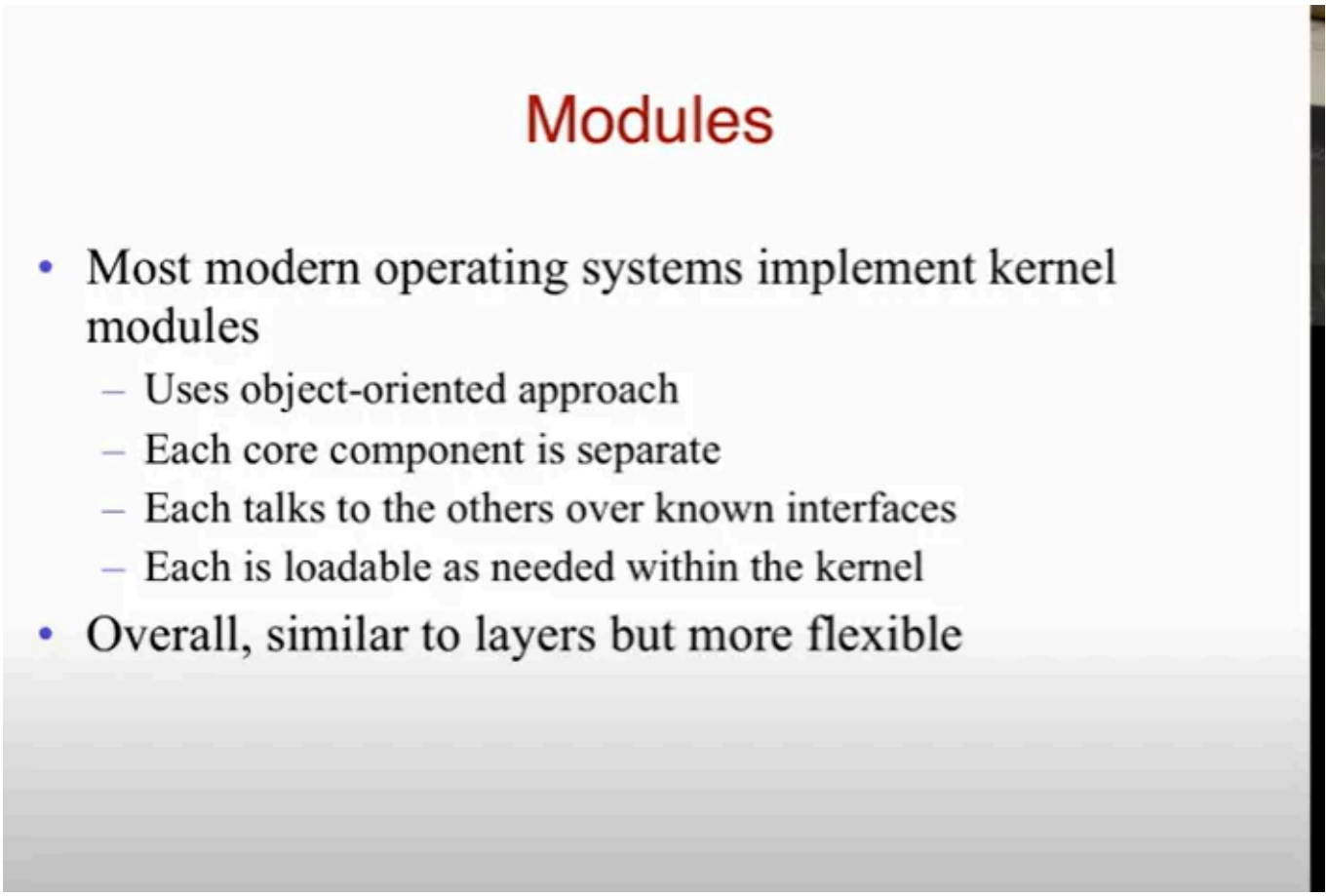
advantages:

- more customizable
- here the kernel is very simple and and lot of functions which are usually in kernel are given to user mode
- overhead is more if the user function need to communicate with the kernel it needs to again and again send syscalls
  and also it makes it reliable because if file system is corrupted the computer wont crash as it is not a part of kernel
  A MONOLITHIC kernel can overcome this because it takes all these function put them into a big single kernel block which harder to customize or debug but saves time in communication between functionalities
  so we can make hybrid kernel to get the best of both worlds
  monlithic - efficiency power

micro-seperation and modularity



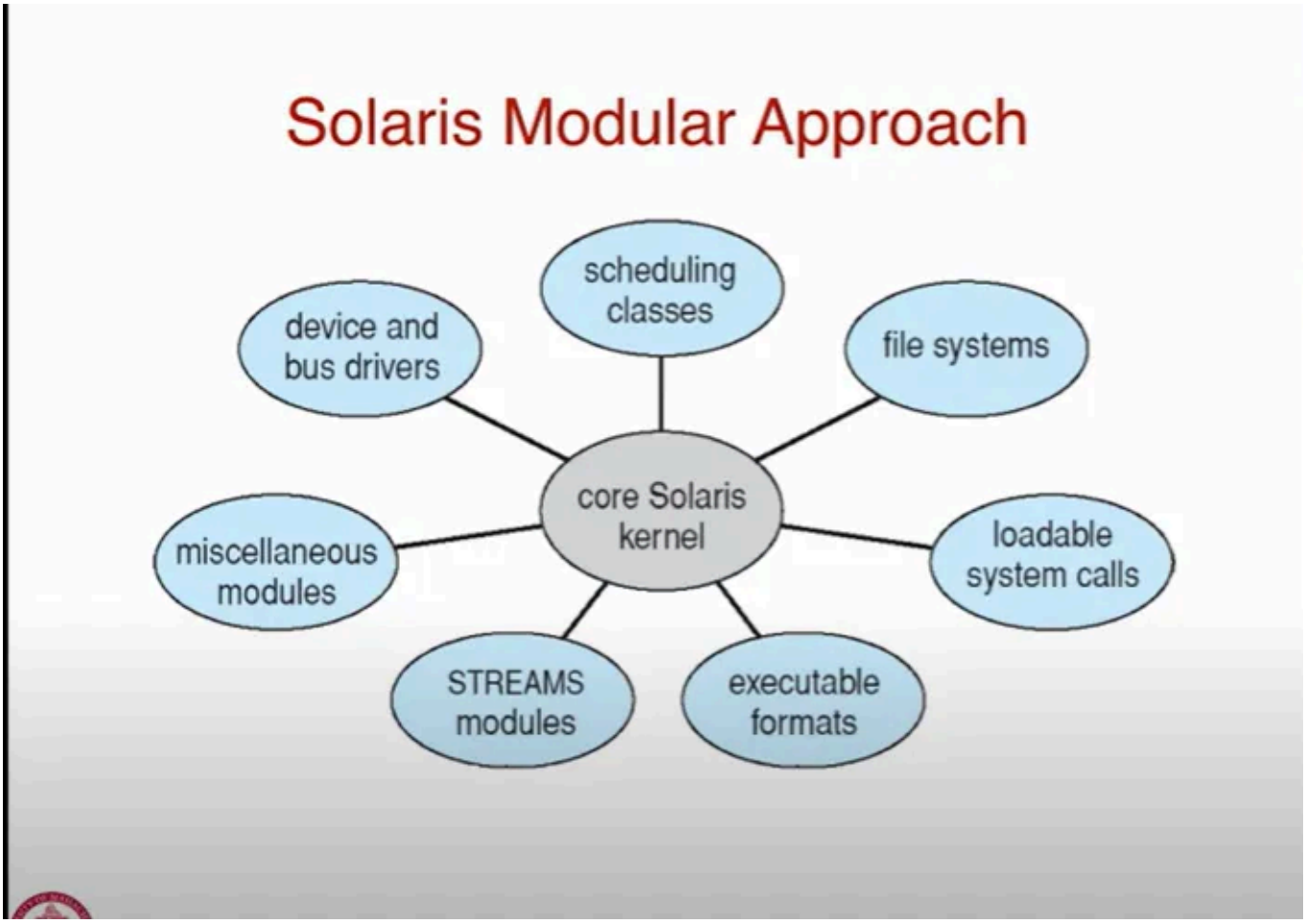
Modules



this hybrid approach can be implemented by dividing the kernel in to modules when the system boots up it starts with the essential modules and adds up the rquired kernel modules as required.

it helps by seperating and the monolithic kernel into parts like a microkernel

and therfore it avaoids recomplilation of entire kernel if we need to add a new functionality just compile the kernel module and add it to the kernel



this doesnt have much security because because even a single corrupted module can cause the whole kernel and system to crash