**Computer Hardware Review**

**1. Processors (CPU)**

* The Central Processing Unit (CPU) is the brain of the computer, responsible for executing instructions.
* Modern CPUs are multi-core, allowing multiple processes to run simultaneously.

**2. Memory**

* **Primary Memory (RAM):** Volatile memory used for temporary data storage while programs are running.
* **Secondary Memory:** Non-volatile storage like Hard Drives (HDDs) and Solid-State Drives (SSDs) for long-term data storage.

**3. Input/Output (I/O) Devices**

* **Input Devices:** Keyboard, mouse, scanner, etc.
* **Output Devices:** Monitor, printer, speakers, etc.
* **Storage Devices:** USB drives, external hard drives, etc.

**4. I/O Bus**

* A communication system that transfers data between components inside a computer, or between computers.
* Common buses include PCI, PCIe, USB, and SATA.

**🧠 Operating System Basics**

**1. Definition**

An Operating System (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.

**2. Architecture of OS**

* **Monolithic Kernel:** All OS services run in kernel space.
* **Microkernel:** Minimal kernel running essential services, with others in user space.
* **Hybrid Kernel:** Combines aspects of monolithic and microkernels.

**🖥️ Types of Operating Systems**

**1. Mainframe OS**

* Designed for large-scale computing operations, handling vast amounts of data and supporting numerous users simultaneously.
* Examples: IBM z/OS, IBM z/VSE.

**2. Server OS**

* Optimized to provide services to other computers over a network.
* Examples: Windows Server, Linux Server distributions.

**3. Multiprocessor OS**

* Supports systems with multiple CPUs, enabling parallel processing.

**4. Embedded OS**

* Designed for embedded systems like appliances, vehicles, and industrial machines.
* Examples: Embedded Linux, VxWorks.

**5. Real-Time OS (RTOS)**

* Processes data as it comes in, typically without buffering delays.
* Examples: RTLinux, FreeRTOS.

**6. Sensor Node OS**

* Lightweight OS designed for sensor networks with limited resources.
* Examples: TinyOS, Contiki.

**7. Smart Card OS**

* Specialized OS for smart cards, focusing on security and minimal resource usage.
* Examples: Java Card OS, MULTOS.

**🧩 Virtualization**

Virtualization allows multiple operating systems to run on a single physical machine by abstracting the hardware.

**Types of Hypervisors**

* **Type 1 (Bare-metal):** Runs directly on hardware.
  + Examples: Xen, KVM.
* **Type 2 (Hosted):** Runs on a host OS.
  + Examples: VirtualBox, VMware Workstation.

**🆓 Free and Open Source Operating Systems**

Open-source OSes are developed collaboratively and are freely available.

* **Linux Distributions:** Ubuntu, Fedora, Debian.
* **BSD Variants:** FreeBSD, OpenBSD, NetBSD.
* **Others:** ReactOS (Windows-compatible), Haiku (BeOS-inspired).

**Operating System Architectures**

**1. Monolithic System**

In a monolithic architecture, the entire OS operates in kernel mode, integrating all essential services like process management, memory management, file systems, and device drivers into a single large process. This design can offer high performance due to minimal inter-process communication overhead. However, it may lack modularity, making maintenance and debugging more challenging.

**2. Layered System**

A layered architecture divides the OS into hierarchical layers, each built upon the one below it. The bottom layer interacts directly with hardware, while the topmost layer interfaces with user applications. This structure enhances modularity and simplifies debugging, as each layer only interacts with its immediate neighbors.

**3. Microkernel**

The microkernel approach minimizes the kernel's responsibilities to essential functions like inter-process communication (IPC), basic scheduling, and memory management. Other services, such as device drivers and file systems, operate in user space. This separation can improve system stability and security, as failures in user-space services are less likely to compromise the entire system.

**4. Client-Server Model**

In this model, the OS is structured as a collection of services (servers) that communicate with client processes. Clients request services, and servers respond, often over IPC mechanisms. This design promotes modularity and can facilitate distributed computing environments.

**🔧 System Components**

**5. System Calls**

System calls provide the interface between user applications and the OS kernel. They allow user programs to request services such as file operations, process control, and communication. System calls are typically accessed through high-level APIs provided by programming languages.

**6. Linker and Loader**

* **Linker**: Combines various object files generated by a compiler into a single executable program. It resolves references between modules and assigns final memory addresses.
* **Loader**: Loads the executable program into memory for execution. It handles memory allocation and sets up the necessary execution environment.

**7. Booting of an Operating System**

Booting is the process of starting a computer and loading the operating system. The sequence typically involves:

1. **Power-On Self-Test (POST)**: Checks hardware components.
2. **BIOS/UEFI Execution**: Initializes system settings and locates the bootloader.
3. **Bootloader Activation**: Loads the OS kernel into memory.
4. **Kernel Initialization**: Sets up the OS environment and starts system processes.

**Process and Threads**

**Process**

A process is a program in execution, representing the basic unit of work in a system. Each process has its own memory space, system resources, and execution context.

**Threads**

Threads are the smallest sequence of programmed instructions that can be managed independently by a scheduler. They exist within processes and share the same memory space.

* **User-Level Threads (ULT):** Managed by user-level libraries; the kernel is unaware of them.
* **Kernel-Level Threads (KLT):** Managed directly by the operating system kernel.

**🔄 Process States and Control**

**Process State Diagram**

A process transitions through various states during its lifecycle:

* **New:** Process is being created.
* **Ready:** Process is waiting to be assigned to a processor.
* **Running:** Instructions are being executed.
* **Waiting:** Process is waiting for some event to occur (like I/O completion).
* **Terminated:** Process has finished execution.

**Process Control Block (PCB)**

The PCB is a data structure maintained by the operating system for every process. It contains:

* **Process State:** Current state of the process.
* **Program Counter:** Address of the next instruction to execute.
* **CPU Registers:** Contents of all process-centric registers.
* **Memory Management Information:** Details like page tables, segment tables.
* **Accounting Information:** CPU usage, process ID, etc.
* **I/O Status Information:** List of I/O devices allocated to the process.

**🗓️ Process Scheduling**

**Scheduling Criteria**

To evaluate scheduling algorithms, several criteria are considered:

**CPU Utilization:** Keep the CPU as busy as possible.

* **Throughput:** Number of processes completed per time unit.
* **Turnaround Time:** Total time taken from submission to completion.
* **Waiting Time:** Total time a process spends waiting in the ready queue.
* **Response Time:** Time from submission to the first response.

**Scheduling Algorithms**

* **First-Come, First-Served (FCFS):** Processes are scheduled in the order they arrive.
* **Shortest Job Next (SJN):** Process with the smallest execution time is selected.
* **Round Robin (RR):** Each process gets a small unit of CPU time in a cyclic order.
* **Priority Scheduling:** Processes are scheduled based on priority.
* **Multilevel Queue Scheduling:** Processes are divided into queues based on priority or type.

**Types of Schedulers**

* **Long-Term Scheduler (Job Scheduler):** Decides which processes are admitted to the system for processing.
* **Short-Term Scheduler (CPU Scheduler):** Decides which of the ready, in-memory processes is to be executed next.
* **Medium-Term Scheduler:** Temporarily removes processes from memory to reduce the degree of multiprogramming.

**🧵 Thread Scheduling**

Thread scheduling can be managed at both user and kernel levels:

* **User-Level Thread Scheduling:** Managed by the thread library; the kernel is unaware of these threads.
* **Kernel-Level Thread Scheduling:** Managed directly by the operating system kernel.

Scheduling policies can be preemptive or non-preemptive, affecting how threads are prioritized and switched.

**🔗 Inter-Process Communication (IPC)**

IPC mechanisms allow processes to communicate and synchronize their actions:

* **Shared Memory:** Multiple processes access a common memory space.
* **Message Passing:** Processes communicate by sending and receiving messages.

These mechanisms are essential for coordinating processes and ensuring data consistency.

**⚠️ Synchronization and Concurrency**

**Race Condition**

Occurs when multiple processes access and manipulate shared data concurrently, and the outcome depends on the sequence of access.

**Critical Section**

A part of the program where shared resources are accessed. Proper synchronization is required to prevent race conditions.

**Synchronization Mechanisms**

* **Semaphore:** An integer variable used to control access to a common resource by multiple processes.
* **Mutex (Mutual Exclusion):** A lock that allows only one thread to access a resource at a time.
* **Monitor:** A high-level synchronization construct that provides a mechanism to control access to shared resources.

**Classical Synchronization Problems**

* **Producer-Consumer Problem:** Ensures that producers don't add data into a full buffer and consumers don't remove data from an empty buffer.
* **Readers-Writers Problem:** Manages access to a resource where readers can access simultaneously, but writers require exclusive access.
* **Dining Philosophers Problem:** Illustrates the challenges of allocating limited resources among multiple processes without deadlock.

**Deadlock Characterization**

A **deadlock** occurs when a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. For a deadlock to occur, the following four Coffman conditions must hold simultaneously:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode; that is, only one process can use the resource at any given time.
2. **Hold and Wait**: A process is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
3. **No Preemption**: Resources cannot be forcibly taken away from a process holding them; they can only be released voluntarily by the process.
4. **Circular Wait**: A set of processes exist such that each process is waiting for a resource held by the next process in the set, forming a circular chain.

If all these conditions are true simultaneously, a deadlock situation arises, leading to potential system failures.

**🛠️ Methods of Handling Deadlocks**

Operating systems employ various strategies to handle deadlocks:

1. **Deadlock Prevention**: This strategy ensures that at least one of the necessary conditions for deadlock cannot hold, thereby preventing the occurrence of deadlocks. For instance, by preventing the "hold and wait" condition, processes must request all required resources at once, reducing the chance of deadlock.
2. **Deadlock Avoidance**: In this approach, the system dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. Algorithms like the Banker's Algorithm are used to allocate resources only if the system remains in a safe state.
3. **Deadlock Detection and Recovery**: Here, the system allows deadlocks to occur but has mechanisms to detect and recover from them. Detection algorithms identify deadlocks, and recovery strategies like terminating processes or preempting resources are employed to resolve the deadlock.
4. **Deadlock Ignorance**: Some systems, like certain versions of UNIX, adopt an approach where they ignore the problem, under the assumption that deadlocks are rare and the cost of prevention is high. This is known as the Ostrich Algorithm.

**🛡️ Deadlock Prevention and Avoidance**

**Deadlock Prevention**

This method involves designing the system in such a way that at least one of the necessary conditions for deadlock cannot occur:

* **Eliminating Mutual Exclusion**: Making resources sharable, though not always possible.
* **Eliminating Hold and Wait**: Requiring processes to request all resources at once, leading to low resource utilization.
* **Eliminating No Preemption**: Allowing the system to preempt resources from processes.
* **Eliminating Circular Wait**: Imposing an ordering on resource acquisition to prevent circular chains.

While prevention strategies can be effective, they may lead to reduced system efficiency and resource utilization.

**Deadlock Avoidance**

Deadlock avoidance requires the system to have additional information about future resource requests. The system makes decisions dynamically to ensure that it never enters an unsafe state. The Banker's Algorithm is a classic example, where the system checks for safe states before allocating resources.

**🔍 Deadlock Detection and Recovery**

**Deadlock Detection**

In systems where deadlocks are allowed to occur, the operating system must have mechanisms to detect them:

* **Resource Allocation Graph (RAG)**: Used when there is only one instance per resource type. A cycle in the graph indicates a deadlock.
* **Wait-For Graph (WFG)**: A simplified version of RAG, used for detecting deadlocks among processes.
* **Detection Algorithms**: Algorithms that check for cycles or unsafe states in the resource allocation.

**Recovery from Deadlock**

Once a deadlock is detected, the system must recover:

* **Process Termination**: Terminating one or more processes involved in the deadlock to break the cycle.
* **Resource Preemption**: Temporarily taking resources away from processes and reallocating them to others.
* **Rollback**: Rolling back processes to a safe state and restarting them.

Each recovery method has its trade-offs between system stability and resource utilization.

**Contiguous Memory Allocation**

In **contiguous memory allocation**, each process is allocated a single contiguous block of memory. This approach simplifies memory management but can lead to fragmentation issues.

* **Fixed Partitioning**: Memory is divided into fixed-size partitions; each partition may contain exactly one process.
* **Variable Partitioning**: Memory is divided dynamically to fit the exact size of incoming processes, which can lead to external fragmentation.

This method is straightforward but can result in inefficient memory utilization due to fragmentation.

**📄 Paging**

**Paging** is a memory management scheme that eliminates the need for contiguous allocation of physical memory. It allows the physical address space of a process to be non-contiguous.

* **Pages and Frames**: The logical memory is divided into blocks of the same size called pages, and the physical memory is divided into blocks of the same size called frames.
* **Page Table**: A data structure used to map virtual addresses to physical addresses.

Paging helps in efficient memory utilization and eliminates external fragmentation.

**🔄 Swapping**

**Swapping** is a technique where processes are moved between main memory and secondary storage (like a hard disk) to manage memory efficiently.

* **Purpose**: To free up memory by moving inactive or less frequently used processes to disk, allowing active processes to execute.
* **Process**: When a process is swapped out, its entire memory image is copied to disk; when it's needed again, it's swapped back into memory.

Swapping increases the degree of multiprogramming but can lead to increased I/O overhead.

**🧠 Virtual Memory**

**Virtual memory** allows the execution of processes that may not be completely in memory. It gives an illusion of a large main memory by using disk space.

* **Demand Paging**: Pages are loaded into memory only when they are needed, reducing memory usage.
* **Page Table Structure**: Maintains the mapping between virtual addresses and physical addresses.
* **Page Replacement Algorithms**: When a page fault occurs, and memory is full, these algorithms decide which page to replace. Common algorithms include:
  + **FIFO (First-In-First-Out)**: Replaces the oldest page in memory.
  + **LRU (Least Recently Used)**: Replaces the page that hasn't been used for the longest time.
  + **Optimal**: Replaces the page that will not be used for the longest time in the future.

Virtual memory enables efficient and safe memory management, allowing systems to run larger applications with limited physical memory.

**⚠️ Page Fault Handling**

A **page fault** occurs when a program tries to access a page that is not currently in memory. The operating system handles this by:

1. Determining the location of the data on disk.
2. Loading the page into a free frame in memory.
3. Updating the page table to reflect the new location.
4. Resuming the execution of the program.

Efficient page fault handling is crucial for system performance.

**📐 Segmentation**

**Segmentation** divides the memory into variable-sized segments based on the logical divisions of a program, such as functions, objects, or data structures.

* **Segment Table**: Each entry contains the base address and the limit (length) of the segment.
* **Advantages**: Provides a more logical view of memory, facilitates sharing and protection, and can reduce fragmentation.

Segmentation aligns with the way programs are structured, making it easier to manage and protect memory.

**File Naming**

File naming conventions vary across operating systems but generally adhere to certain rules:

* **Structure**: A file name typically consists of two parts: the name and an extension, separated by a period (e.g., report.docx).
* **Length and Characters**: Many file systems support names up to 255 characters, including letters, numbers, and certain special characters.
* **Case Sensitivity**: Some systems, like UNIX, are case-sensitive (File.txt and file.txt are different), while others, like Windows, are not.

These conventions help in organizing and accessing files efficiently.

**🗂️ File Structures and Types**

Files can be organized in various structures, depending on the operating system:

* **Flat (Unstructured) Files**: A sequence of bytes with no inherent structure.
* **Structured Files**: Contain records or other structures, facilitating easier data management.

Common file types include:

* **Text Files**: Contain readable characters.
* **Binary Files**: Contain data in binary format, not directly readable.
* **Executable Files**: Contain programs that can be run by the operating system.

Understanding file structures and types is crucial for proper file handling and processing.

**🗃️ Directory Structures: Single-Level and Hierarchical**

Directory structures help in organizing files systematically:

* **Single-Level Directory**: All files are stored in a single directory. Simple but can lead to naming conflicts and difficulties in managing large numbers of files.
* **Hierarchical (Tree-Structured) Directory**: Files are organized in a tree-like structure with directories and subdirectories, allowing for better organization and scalability.

Hierarchical structures are more common in modern operating systems due to their flexibility and efficiency.

**🔗 Shared Files**

In multi-user systems, sharing files is essential:

* **Access Control**: Permissions are set to control who can read, write, or execute a file.
* **Concurrent Access**: Mechanisms like file locking prevent conflicts when multiple users access the same file simultaneously.

Proper management ensures data integrity and security in shared environments.

**💽 Disk Management and Related Algorithms**

Disk management involves organizing and accessing data on storage devices efficiently:

* **Disk Scheduling Algorithms**: Determine the order in which disk I/O requests are processed. Common algorithms include:
  + **First-Come, First-Served (FCFS)**: Processes requests in the order they arrive.
  + **Shortest Seek Time First (SSTF)**: Selects the request closest to the current head position.
  + **SCAN (Elevator Algorithm)**: Moves the disk arm in one direction, servicing requests, then reverses direction.
  + **C-SCAN (Circular SCAN)**: Similar to SCAN but returns to the beginning without servicing requests on the return trip.

These algorithms aim to reduce seek time and improve overall system performance.

**Security Threats and Attackers**

Operating systems face various security threats from different types of attackers:

* **External Threats**: These include cybercriminals, nation-state actors, and hacktivists who exploit vulnerabilities to gain unauthorized access or disrupt services. For instance, the UK has experienced a significant increase in cyberattacks, with 200 incidents recorded since September, doubling the number from the previous year.
* **Internal Threats**: Insiders, such as disgruntled employees or contractors, may misuse their access privileges to compromise system security.
* **Zero-Day Exploits**: These are attacks that target previously unknown vulnerabilities, leaving systems unprotected until a patch is released.

**🛡️ Controlling Access to Resources**

Effective access control mechanisms are vital for protecting system resources:

**Protection Domain**

A **protection domain** defines a set of objects (e.g., files, devices) and the types of operations (e.g., read, write) that subjects (e.g., users, processes) can perform on them. Each domain enforces specific access rights, ensuring that subjects can only interact with objects in authorized ways .

**Access Control List (ACL)**

An **Access Control List (ACL)** is a list associated with a resource that specifies which users or system processes have access to that resource and what operations they can perform. For example, an ACL for a file might grant read and write permissions to one user and read-only permissions to another.

**🧑‍💻 Exploiting Software Vulnerabilities**

Attackers often exploit software vulnerabilities to compromise systems:

**Buffer Overflow Attack**

A **buffer overflow** occurs when a program writes more data to a buffer than it can hold, potentially overwriting adjacent memory. This can lead to arbitrary code execution or system crashes. There are two main types:

* **Stack-based Buffer Overflow**: Overflows occur in the call stack, potentially overwriting return addresses .
* **Heap-based Buffer Overflow**: Overflows occur in the heap memory area, affecting dynamically allocated memory.

**Integer Overflow Attack**

An **integer overflow** happens when an arithmetic operation results in a value outside the range that can be represented with a given number of bits. Attackers can exploit this to bypass security checks or cause unexpected behavior.

**Dangling Pointer**

A **dangling pointer** refers to a pointer that continues to reference a memory location after the object it points to has been deallocated. Dereferencing such pointers can lead to undefined behavior, including crashes or security vulnerabilities.

**Null Pointer Dereference Attack**

A **null pointer dereference** occurs when a program attempts to read or write to a memory location through a pointer that has been set to null. This can cause system crashes and may be exploited to execute arbitrary code.

**🦠 Malware: Worms, Viruses, and Trojans**

Malware is malicious software designed to damage or exploit systems:

**Virus**

A **virus** attaches itself to legitimate programs or files and spreads when these are executed or opened. It often requires user action to propagate.

**Worm**

A **worm** is a standalone program that replicates itself to spread to other computers, often exploiting network vulnerabilities. Unlike viruses, worms do not need to attach to existing programs.

**Trojan**

A **Trojan horse** disguises itself as legitimate software to trick users into installing it. Once activated, it can perform malicious actions such as stealing data or creating backdoors.

**Goals of Distributed Operating Systems**

A Distributed Operating System manages a collection of independent computers, making them appear to users as a single coherent system. The primary goals include:

* **Transparency**: Hiding the complexities of the distributed nature from users. This includes:
  + *Access Transparency*: Users access resources without knowing their physical location.
  + *Location Transparency*: Resources can be accessed without knowledge of their location.
  + *Replication Transparency*: Users are unaware of multiple copies of resources.
  + *Concurrency Transparency*: Multiple users can access resources concurrently without interference.
  + *Failure Transparency*: System continues to function despite failures.
* **Scalability**: System performance remains efficient as the number of users and resources increases.
* **Reliability and Fault Tolerance**: The system continues to operate correctly even in the presence of hardware or software failures.
* **Resource Sharing**: Efficient sharing of resources like files, printers, and processing power among multiple users.
* **Performance**: Optimizing the system to provide high throughput and low response time.

**🔁 Remote Procedure Call (RPC)**

**Remote Procedure Call (RPC)** is a protocol that allows a program to execute a procedure on another address space (commonly on another physical machine) as if it were a local procedure call.

**How RPC Works:**

1. **Client Stub**: The client calls a local stub procedure, which packages the procedure parameters into a message.
2. **Communication**: The message is sent over the network to the server.
3. **Server Stub**: The server stub receives the message, unpacks the parameters, and calls the desired procedure on the server.
4. **Execution and Response**: After execution, the result is sent back through the server stub to the client stub, which unpacks it and returns it to the client program.

**Advantages:**

* Simplifies the process of executing code across different systems.
* Abstracts the complexities of network communication.

**🧭 Name Resolution in Distributed Systems**

In distributed systems, **name resolution** is the process of mapping a human-readable name to a system resource or address. This is crucial for locating resources across the network.

**Key Concepts:**

* **Naming Systems**: Maintain mappings between names and resources. Examples include DNS (Domain Name System) for resolving domain names to IP addresses.
* **Name Servers**: Specialized servers that handle name resolution requests.
* **Binding**: Associating a name with a specific resource or service.

**⏰ Clock Synchronization**

In distributed systems, each node may have its own clock, leading to discrepancies in timekeeping. **Clock synchronization** ensures a consistent time across all nodes, which is vital for coordination and consistency.

**Techniques:**

* **Network Time Protocol (NTP)**: Synchronizes clocks of computer systems over packet-switched, variable-latency data networks. It can achieve accuracy in the range of milliseconds.
* **Precision Time Protocol (PTP)**: Provides higher precision time synchronization (within microseconds) suitable for systems requiring precise timing, such as high-frequency trading platforms.
* **Cristian's Algorithm**: A method where a client synchronizes its clock with a time server by accounting for message transmission delays.
* **Berkeley Algorithm**: Averages the time of all nodes to synchronize clocks, suitable for systems without an authoritative time source.

**Importance:**

* Maintains the order of events across the system.
* Essential for time-sensitive applications and consistency in distributed databases.