The kernel is an important part of an Operating System.

The kernel is the first program that is loaded after the boot loader (A **boot loader**, also spelled as boot loader or called **boot manager** and **bootstrap loader**, is a computer program that is responsible for booting a computer) whenever we start a system.

The Kernel is present in the memory until the Operating System is shut-down.

The kernel provides an interface between the user and the hardware components of the system. When a process makes a request to the Kernel, then it is called System Call.

Functions

The functions of the kernel are as follows −

Process management

Access computer resources

Device management

Memory management

Interrupt handling

I/O communication

File system...etc.

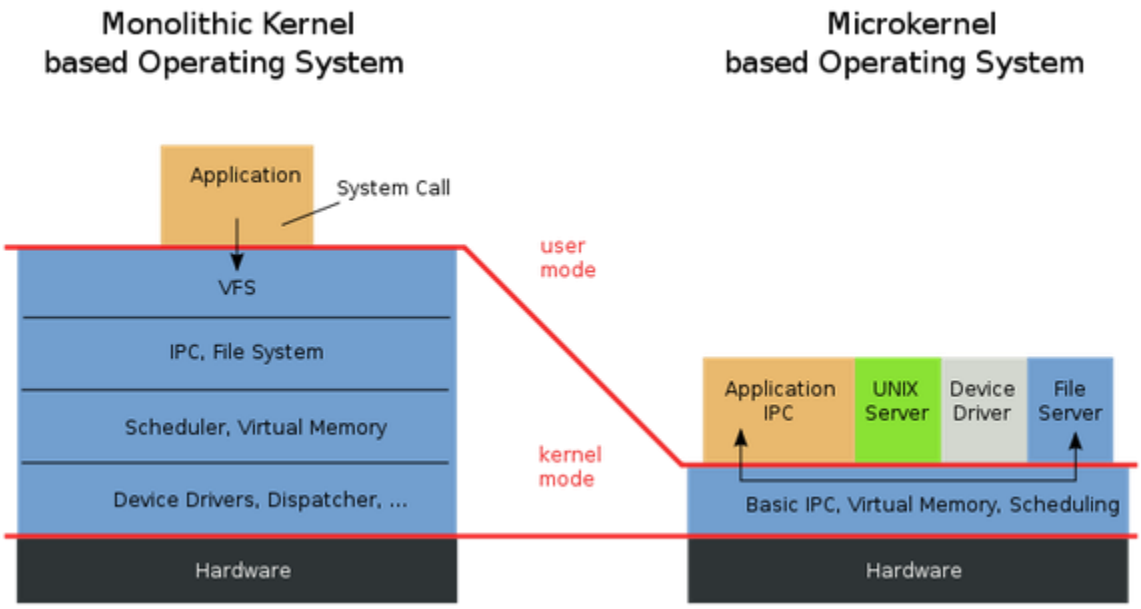
Access Computer resource − A Kernel accesses various computer resources like the CPU, I/O devices, and other resources. The kernel is present between the user and the resources of the system to establish communication.

Resource Management − Kernel shares the resources between various processes in a way that there is uniform access to the resources by every process.

Memory Management − Generally memory management is done by the kernel because every process needs some memory space and memory has to be allocated and deallocated for its execution.

Device Management − The allocation of peripheral devices connected in the system used by the processes is managed by the kernel.

Types of Kernel



The different types of kernels are as follows −

Monolithic Kernels

In monolithic Kernels, both user services and kernel services are implemented in the same memory space.

By doing this, the size of the Kernel is increased and at the same time, it increases the size of the Operating System.

As there is no separate User Space and Kernel Space, so the execution of the process will be faster in Monolithic Kernels.

Advantages

It provides CPU scheduling, memory scheduling, file management through System calls.

Execution of the process is fast as there is no separate space.

Disadvantages

If the service fails, then the system failure happens.

If you try to add new services then the entire Operating System needs to be modified.

Microkernel

A Microkernel is not the same as a Monolithic kernel. It is a little bit different because, in a Microkernel, the user services and kernel services are implemented into different spaces.

Because of using User Space and Kernel Space separately, it reduces the size of the Kernel and in turn, reduces the size of the Operating System.

As we are using different spaces for user and kernel service, the communication between application and services is done with the help of message parsing because of this it reduces the speed of execution.

The *advantage* of a microkernel is that it can easily add new services at any time.

The *disadvantage* of a microkernel is that here we are using User Space and Kernel Space separately. So, the communication between these can reduce the overall execution time.

Hybrid Kernel

It is the combination of both a Monolithic Kernel and a Microkernel. It uses the speed of the Monolithic Kernel and the modularity of the Microkernel.

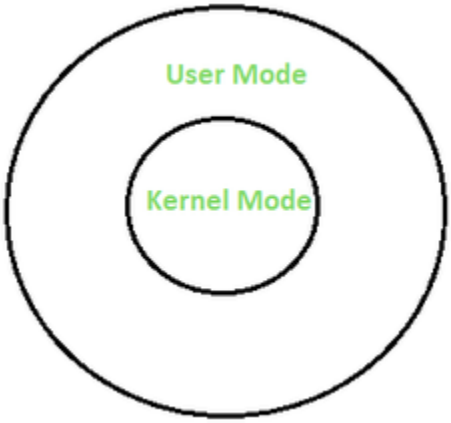
Hybrid kernels are micro kernels having some non-essential code in kernel space in order for the code to run more quickly than it would in user space. So, some services like network stack or file systems are run in Kernel space to reduce the performance overhead, but still, it runs kernel code as servers in the user space.

Nanokernel

The name suggests the complete code of the kernel is very small, which means the code executing in the privileged mode of the hardware is very small. The term nanokernel is used to explain that the kernel supports a nanosecond clock resolution.

Exokernel

Exokernel is an Operating System kernel developed by the group of MIT parallel and Distributed Operating Systems. In this type of kernel, the resource protection is separated from the management which results in allowing us to perform application-specific customization.



Criteria	Kernel Mode (Master mode, privileged mode, or system mode)	User Mode (Unprivileged mode, restricted mode, or slave mode)
Access to Resources	the program has direct and unrestricted access to system resources.	the application program does not have direct access to system resources. In order to access the resources, a system call must be made.
Interruptions	the whole operating system might go down if an interruption occurs	In user mode, a single process fails if an interrupt occurs.
Virtual address space	all processes share a single virtual address space.	all processes get separate virtual address space.
Level of privilege	the applications have more privileges as compared to user mode.	While in user mode the applications have fewer privileges.

System Call

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a system call.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a context switch.

Then the kernel provides the resource which the program requests. After that, another context switch happens which results in a change of mode from kernel mode back to user mode.

Generally, system calls are made by the user-level programs in the following situations:

- Creating, opening, closing, and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network.

A. Define an operating system and discuss its role from different perspectives (2020, 2021)

An operating system (OS) is software that acts as an intermediary between computer hardware and user applications. It manages computer resources, provides an environment for software execution, and enables users to interact with the computer system. Here's a discussion of the role of an operating system from different perspectives:

❖ Resource Management:

Processor Management: The OS allocates processor time to different processes or threads, ensuring efficient and fair execution.

Memory Management: It controls the allocation and deallocation of memory to running programs, optimizing memory usage and preventing conflicts.

Device Management: The OS facilitates communication between software and hardware devices, handling input/output operations and device drivers.

File System Management: It organizes and controls access to files and directories, providing a hierarchical structure and ensuring data integrity.

❖ User Interface:

Command-Line Interface (CLI): The OS offers a text-based interface where users can input commands to perform tasks.

Graphical User Interface (GUI): It provides a visual interface with icons, windows, and menus, allowing users to interact with the system through pointing and clicking.

- **Process and Task Management:**

Process Scheduling: The OS determines the order and priority of executing processes, maximizing CPU utilization and responsiveness.

Multitasking: It allows concurrent execution of multiple processes, providing the illusion of parallelism and enabling users to run multiple applications simultaneously.

Process Communication: The OS facilitates communication and data exchange between processes through inter-process communication mechanisms.

- **Security and Protection:**

User Authentication: The OS verifies user identities to ensure secure access to the system, protecting sensitive data and resources.

Authorization: It controls user permissions, granting or restricting access to files, folders, and system resources based on predefined rules.

Data Protection: The OS implements security measures like encryption, access control, and firewall to safeguard data from unauthorized access and malicious activities.

- **Error Handling and Fault Tolerance:**

Error Detection and Reporting: The OS identifies and reports errors in software or hardware components, helping in debugging and troubleshooting.

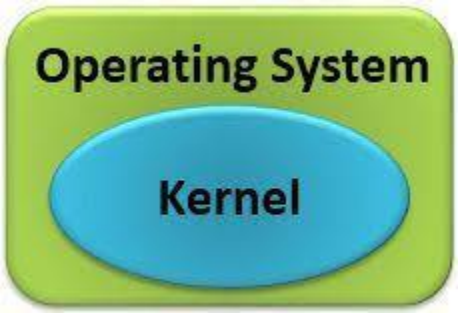
Fault Tolerance: It provides mechanisms to handle system failures, such as backup and recovery procedures, to minimize data loss and system downtime.

- **System Performance:**

Resource Monitoring: The OS tracks resource utilization, including CPU, memory, disk usage, and network activity, assisting in system optimization and performance analysis.

Performance Optimization: It employs techniques like caching, buffering, and prefetching to enhance system responsiveness and efficiency.

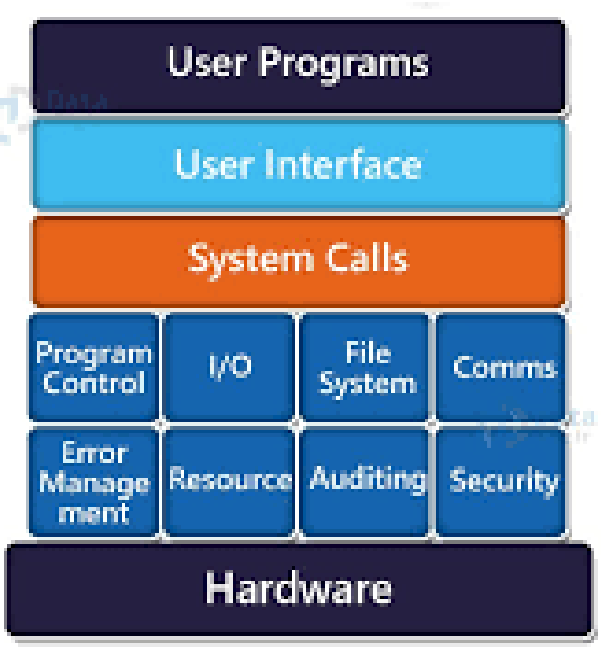
Discuss the importance of the kernel in the Linux operating system. (2020)



The kernel is a part of the operating system that converts user commands into machine language.

the kernel is the foundational component of the Linux operating system, responsible for managing resources, enabling hardware interaction, ensuring security, and providing stability. Its importance lies in its ability to abstract the underlying hardware, facilitate efficient resource utilization, and offer a customizable and secure environment for running software applications.

C. What is the purpose of the system calls? (2020, 2021)



A system call is a function that a user program uses to ask the operating system for a particular service. User programmers can communicate with the operating system to request its services using the interface that is created by a system call.

Accessing hardware resources:

System calls enable user processes to communicate with hardware devices such as disks, printers, network interfaces, and input/output devices. They provide an abstraction layer that allows programs to issue commands, read from or write to hardware, and handle device-related operations.

File system operations:

System calls provide mechanisms to manage files and directories. They allow processes to create, read, write, delete, and manipulate files, as well as perform operations like file opening, closing, and seeking.

Process management:

System calls facilitate the creation, termination, and control of processes. They enable processes to request resources, fork (create a new process), execute programs, manage process attributes (such as priority and permissions), and communicate with other processes.

Memory management:

System calls allow processes to allocate and deallocate memory dynamically. They enable processes to request memory from the operating system, map files into memory, share memory with other processes, and handle memory protection and permissions.

Interprocess communication (IPC):

System calls provide mechanisms for processes to communicate and synchronize with each other. They allow processes to send messages, share data, and coordinate their activities using techniques like shared memory, pipes, sockets, signals, and semaphores.

Network operations:

System calls offer interfaces for network communication. They enable processes to establish network connections, send and receive data over networks, perform network-related configurations, and interact with protocols like TCP/IP and UDP.

Security and access control:

System calls provide means to enforce security policies and regulate access to system resources. They allow processes to authenticate, manage user accounts, set permissions on files and directories, and handle encryption and decryption operations.

D. Why do we use APIs rather than system calls? (2021)

While system calls provide a direct interface between user-level processes and the operating system, APIs (Application Programming Interfaces) offer a higher-level abstraction, portability, modularity, and functionality compared to system calls which makes them preferable in many scenarios. They simplify development, enhance code reuse, and provide optimized access to system resources, making them a preferred choice for building applications and software systems.

Abstraction and Simplification:

APIs provide a simplified and more intuitive interface compared to system calls. They abstract away the low-level details of system operations, making it easier for developers to interact with complex functionalities of the operating system or other software components.

Portability:

APIs offer a portable way to develop applications that can run on different platforms and operating systems. By using APIs, developers can write code that is independent of the underlying system calls or implementation details, allowing their software to be more easily adapted to various environments.

Modularity and Code Reusability:

APIs promote modularity and code reusability by encapsulating functionality into reusable components. Developers can use APIs to access specific features or services without needing to understand the underlying system calls or implement them from scratch. This speeds up development, reduces errors, and promotes code organization.

Performance and Optimization:

APIs can provide optimized and efficient implementations of common operations. The underlying system calls may involve additional overhead or system-specific considerations that are not exposed to the application. APIs can leverage optimizations and performance improvements specific to the platform to enhance the execution speed and efficiency of the code.

Higher-level Functionality:

APIs often provide additional higher-level functionality beyond what is offered by system calls alone. They can include libraries, frameworks, or services that offer advanced features, such as graphical user interfaces, data manipulation, networking protocols, database access, and more. APIs abstract these functionalities into easy-to-use interfaces, enabling developers to leverage powerful capabilities without dealing with low-level details.

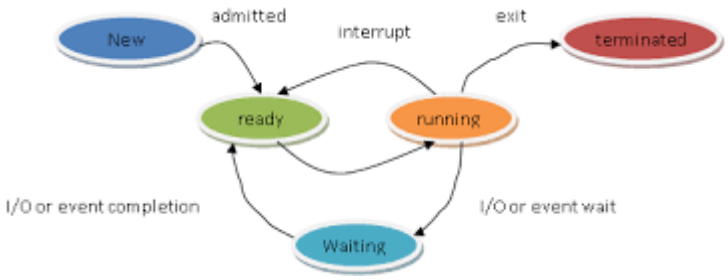
Security and Stability:

APIs provide a controlled and secure way for applications to access system resources. They often incorporate access control mechanisms, input validation, error handling, and security features that help protect the system and prevent misuse or unauthorized access.

Ecosystem and Developer Support:

APIs are typically well-documented and supported by a broader developer community. They often come with comprehensive documentation, examples, and libraries that facilitate development and foster collaboration. Additionally, APIs may have tools and utilities that aid in debugging, testing, and integration with other software components.

B. **Draw a process state diagram and briefly describe each state and the transitions that occur between them. (2020, 2021)**



- **Ready (R):**

This is the initial state of a process when it is loaded into memory and waiting to be executed by the CPU. In this state, the process is prepared to run, but it is waiting for the CPU to be available.

The transition from this state occurs when the CPU scheduler selects the process for execution.

- **Running (X):**

When a process is running, it means that it is currently being executed by the CPU. The process is actively using system resources and performing its tasks.

The transition from this state occurs when the process voluntarily releases the CPU (for example, when it completes its execution) or when it is interrupted by an external event (such as an interrupt request or a higher-priority process becoming ready).

- **Blocked (B):**

This state represents a process that cannot continue its execution because it is waiting for some event or resource. For example, if a process needs to read data from a file or wait for user input, it may enter the blocked state until the required resource becomes available.

The transition from this state occurs when the awaited event or resource becomes available, allowing the process to move back to the ready state.

The following events may occur in a process. Identify the starting state at the time of the event and the ending state. (2020)

	starting state	ending state
Scheduler dispatches process	Ready / waiting	running
I/O complete	Waiting	Ready
Process admitted to ready queue for the first time	New	Ready
Scheduler preempts (interrupts)	Running	ready
The process finishes execution (the task is complete)	Running	terminated

E. When a process creates a new process using the fork() operation, which are the states among stack, heap, and shared memory segments that are shared between the parent process and the child process? (2021)

When a process creates a new process using the fork() operation, certain segments of memory are shared between the parent process and the child process. These shared segments include:

Code Segment (Text Segment): The code segment contains the executable instructions of the program. When a process forks, the child process shares the same code segment as the parent. This sharing is possible because the code segment is typically read-only and does not require modifications.

Shared Libraries: If the parent process has loaded shared libraries (e.g., dynamically linked libraries) during its execution, the child process shares those libraries as well. Shared libraries provide common code that multiple processes can use, resulting in memory efficiency.

Shared Data (Initialized and Uninitialized Static Data): The child process shares the initialized and uninitialized static data segments with the parent process. This includes global variables and static variables defined within the program.

File Descriptors: File descriptors, which represent open files or network connections, are typically shared between the parent and child processes. Both processes can access and operate on the same files or network connections using these shared file descriptors.

Shared Memory: If the parent process has explicitly created shared memory segments using interprocess communication mechanisms like shmget() and map (), those shared memory segments are accessible to the child process as well. Shared memory allows efficient and direct communication between the parent and child processes by sharing a common memory region.

It's important to note that while these segments are shared initially after the fork() operation, subsequent modifications to these shared segments in either the parent or child process can result in separate and independent copies of those segments.

On the other hand, certain segments are not shared between the parent and child processes:

Stack Segment: The stack segment contains the function call stack, including local variables and function call information. When a fork() occurs, the stack segment is typically not shared between the parent and child processes. Instead, each process receives a separate copy of the stack.

Heap Segment: The heap segment is used for dynamic memory allocation, such as using malloc() or new programming languages. Like the stack segment, the heap is not shared between the parent and child processes. Each process maintains its own separate heap space.

F. #include <stdio.h>

#include <unistd.h>

int main() {

fork();

fork();

fork();

return 0; }

including the initial parent process, how many processes are created by the program

Initially, the original process will execute the first fork() call. This will create a new child process, resulting in two processes.

Both the original process and the first child process will now execute the second fork() call. Each process will create another child process, resulting in a total of four processes (two from the original process and two from the first child process).

Now, all four processes will execute the third fork() call. Each process will create another child process, resulting in a total of eight processes.

In total, including the initial parent process, the program will create eight processes.

G. Program VS process

Program: A program is a set of instructions written in a programming language that specifies a sequence of operations to be performed. Programs are not actively executing and require an execution environment, such as an operating system, to run.

Process: A process is an instance of a program that is currently being executed. It is an active entity and represents the execution of a program. When a program is loaded into memory and executed, it becomes a process.

Process VS thread

Process: A process is an instance of a program that is being executed. It is a self-contained unit with its own memory space, resources, and a unique process identifier (PID). Each process runs independently of other processes and can consist of multiple threads.

Thread: A thread is a lightweight unit of execution within a process. It shares the same memory space and resources as other threads within the same process. Threads are often referred to as "lightweight processes" because they are more efficient to create and switch between than processes.

H. Ready State VS Running State

Ready State: In the ready state, a process is prepared to execute but is waiting for the CPU to be allocated to it by the operating system. The process is loaded into main memory and is eligible to run, but it is not currently executing.

Running State: In the running state, a process is currently being executed by the CPU. The operating system has assigned the CPU to the process, and the process is actively executing its instructions.

I. Preemptive Scheduling VS Non-Preemptive Scheduling

Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

Non-Preemptive Scheduling **(** cooperative scheduling or voluntary scheduling):

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non-preemptive scheduling does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then it can allocate the CPU to another process.

J. Long-term scheduler VS Short term scheduler

A long-term scheduler is an operating system scheduler that chooses processes from the job queue and loads them to execution in the main memory.

a short-term scheduler is an operating system scheduler that chooses the process from the several processes that the processor runs.

K. Define the following scheduling criteria: CPU utilization, throughput, turnaround time, waiting time, and response time. for each metric, state whether the goal is to minimize or maximize the metric

- **CPU Utilization:**

CPU utilization refers to the percentage of time the CPU is busy executing a process rather than being idle.

The goal is to maximize CPU utilization, as higher utilization indicates efficient usage of the CPU resources.

- **Throughput:**

Throughput is the total number of processes that are completed within a given period.

The goal is to maximize throughput, as a higher value indicates a greater number of processes being executed and completed in a given time frame.

- **Turnaround Time:**

Turnaround time is the total time taken from the submission of a process to its completion, including both execution time and waiting time.

The goal is to minimize turnaround time, as a shorter turnaround time indicates faster process completion and better overall system performance.

- **Waiting Time:**

Waiting time is the total amount of time a process spends in the ready state, waiting to be allocated the CPU for execution.

The goal is to minimize waiting time, as shorter waiting times indicate faster responsiveness and better utilization of CPU resources.

- **Response Time:**

Response time is the time elapsed between submitting a request or initiating a process and receiving the first response or output.

The goal is to minimize response time, as shorter response times lead to quicker system feedback and better user experience.

What is a context switch?

The Context switching is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system.

Name and describe 3 CPU scheduling algorithms.

First-Come, First-Served (FCFS) Scheduling:

In FCFS scheduling, the CPU is assigned to the process that arrives first and is placed in the ready queue. The processes are executed in the order of their arrival time.

Shortest Job Next (SJN) Scheduling (also known as Shortest Job First, or SJF):

Description: SJN scheduling selects the process with the smallest burst time (execution time) first for execution. It prioritizes the shortest job to minimize the average waiting time.

Round Robin (RR) Scheduling:

Description: RR scheduling assigns a fixed time quantum (time slice) to each process in the ready queue. The CPU executes a process for the time quantum and then switches to the next process in the queue, following a circular fashion.

L.

Process	Burst Time	Priority
P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFD, SJF, non-preemptive priority(a larger priority number implies a higher priority), and RR (quantum=2).

1. First-Come, First-Served (FCFS):

| P1 | P2 | P3 | P4 | P5 |

2. Shortest Job First (SJF):

| P2 | P1 | P4 | P5 | P3 |

3. Non-preemptive Priority Scheduling:

| P3 | P5 | P1 | P4 | P2 |

4. Round-Robin Scheduling (Quantum = 2):

M. Turnaround Time = Completion Time - Arrival Time

Waiting time = Turn Around time – Burst time

calculate the turnaround time of these processes Using FCFS scheduling algorithms

Process	Burst Time	Arrival Time	Completion Time	Turnaround Time
P1	2	0	2	2
P2	1	0	3	3
P3	8	0	11	11
P4	4	0	15	15
P5	5	0	20	20

calculate the turnaround time of these processes Using SJF scheduling algorithms

Process	Burst Time	Arrival Time	Completion Time	Turnaround Time
P1	2	0	3	3
P2	1	0	1	1
P3	8	0	20	20
P4	4	0	7	7
P5	5	0	12	12

calculate the turnaround time of these processes Using Non-preemptive Priority Scheduling algorithms

Process	Burst Time	Arrival Time	Completion Time	Turnaround Time
P1	2	0	15	15
P2	1	0	20	20
P3	8	0	8	8
P4	4	0	19	19
P5	5	0	13	13

calculate the turnaround time of these processes Using Round-Robin Scheduling (Quantum = 2) Scheduling algorithms

Process	Burst Time	Arrival Time	Completion Time	Turnaround Time
P1	2	0		
P2	1	0		
P3	8	0		
P4	4	0		
P5	5	0		

N. Four necessary conditions for deadlock

Mutual exclusion - Each resource is either currently allocated to exactly one process or is available. (Two processes cannot simultaneously control the same resource or be in their critical section).

Hold and Wait - processes currently holding resources can request new resources

No preemption - Once a process holds a resource, it cannot be taken away by another process or the kernel.

Circular wait - Each process is waiting to obtain a resource that is held by another process.

O. State a simple rule for avoiding deadlocks

One simple rule for avoiding deadlocks is the "Resource Ordering" rule. According to this rule, we can assign a partial order to the resources and require that each process requests resources in a specific order.

By enforcing a consistent order for resource allocation, we can prevent circular wait, one of the necessary conditions for deadlock. This rule ensures that processes always acquire resources in a predetermined sequence, eliminating the possibility of a circular dependency.

For example, if we have resources A, B, and C, we can define the resource ordering as A < B < C. This means that a process must acquire resource A before requesting resource B, and it must acquire resource B before requesting resource C. By following this rule, we ensure that processes do not create circular dependencies when acquiring resources, thereby avoiding deadlocks.

P.

	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

What is the content of the matrix need?

Need = Max – Allocation

the matrix "need

	A	B	C	D
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

Q. Is this system in a safe state?

Work = available = 1 5 2 0

Finish [i] = False : i=0,1,2,3,4 i = process

False	False	false	false	False

Need <= Work

If we consider the P0 process (i=0)

Need <= Work

0 0 0 0 <= 1 5 2 0 ----------→ true

So new work = current work + allocation[i]

= 1 5 2 0 + 0 0 1 2

work = 1 5 3 2

Finish[0] = true

True	False	false	false	False

If we consider the P0 process (i=1)

Need <= Work

0 7 5 0 <= 1 5 3 2 ----------→ false

So p1 must wait

True	False	false	false	False

If we consider the P0 process (i=2)

Need <= Work

1 0 0 2 <= 1 5 3 2 ----------→ true

So new work = current work + allocation[i]

= 1 5 3 2 + 1 3 5 4

work = 2 8 8 6

Finish[2] = true

True	False	true	false	False

If we consider the P0 process (i=3)

Need <= Work

0 0 2 0 <= 2 8 8 6----------→ true

So new work = current work + allocation[i]

= 2 8 8 6+ 0 6 3 2

work = 2 14 11 8

Finish[3] = true

True	False	true	true	False

If we consider the P0 process (i=4)

Need <= Work

0 6 4 2 <= 2 14 11 8----------→ true

So new work = current work + allocation[i]

= 2 14 11 8 + 0 0 1 4

work = 2 14 12 12

Finish[4] = true

True	False	true	true	True

If we consider the P0 process (i=1)

Need <= Work

0 7 5 0 <= 2 14 12 12----------→ true

So new work = current work + allocation[i]

= 2 14 12 12 + 1 0 0 0

work = 3 14 12 12

Finish[1] = true

True	True	true	true	True

P0, p2, p3, p4, p1 = true, true, true , true, true

So this system is in the safe state

R. If a request from process p1 arrives for (0,4,2,0) can the request be granted immediately?

To determine if the request from Process P1 for (0, 4, 2, 0) can be granted immediately, we need to check if it satisfies the safety and resource availability conditions.

Available resources: A B C D (1 5 2 0)

if the request (0, 4, 2, 0) for P1 can be granted immediately:

Check if the request is less than or equal to the available resources: (0 4 2 0) <= (1 5 2 0).

The request satisfies the resource availability condition.

So update matrix

	Allocation	Max	Available	Need
	A B C D	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0 - 0 4 2 0 = 1 1 0 0	0 0 0 0
P1	1 0 0 0 + 0 4 2 0 = 1 4 2 0	1 7 5 0		0 3 3 0
P2	1 3 5 4	2 3 5 6		1 0 0 2
P3	0 6 3 2	0 6 5 2		0 0 2 0

P4	0 0 1 4	0 6 5 6		0 6 4 2

If we consider the P0 process (i=0)

Need <= Work

0 0 0 0 <= 1 1 0 0 ----------→ true

So new work = current work + allocation[i]

= 1 1 0 0+ 0 0 1 2

work = 1 1 1 2

Finish[0] = true

True	False	False	false	False

If we consider the P0 process (i=2)

Need <= Work

1 0 0 2 <= 1 1 1 2----------→ true

So new work = current work + allocation[i]

= 1 1 1 2 + 1 3 5 4

work = 2 4 6 6

Finish[2] = true

True	False	True	false	False

If we consider the P0 process (i=3)

Need <= Work

0 0 2 0 <= 2 4 6 6 ----------→ true

So new work = current work + allocation[i]

= 2 4 6 6 + 0 6 3 2

work = 2 10 9 8

Finish[3] = true

True	False	True	true	False

If we consider the P0 process (i=4)

Need <= Work

0 6 4 2 <= 2 10 9 8----------→ true

So new work = current work + allocation[i]

= 2 10 9 8 + 0 0 1 4

work = 2 10 10 12

Finish[4] = true

True	False	True	true	True

If we consider the P0 process (i=1)

Need <= Work

0 3 3 0 <= 2 10 10 12 ----------→ true

So new work = current work + allocation[i]

= 2 10 10 12 + 1 4 2 0

work = 3 14 12 12

Finish[1] = true

True	True	true	true	True

P0, p2, p3, p4, p1 = true, true, true , true, true

All processes have been marked as complete. Therefore, the system is in a safe state.

So The request from Process P1 for (0, 4, 2, 0) can be granted immediately without resulting in an unsafe state.

	Allocation	**Max**	**Available**

	A B C D E	A B C D E	A B C D E
P0	1 0 2 1 1	1 1 2 1 1	0 0 x 1 1
P1	2 0 1 1 0	2 2 2 1 0	
P2	1 1 0 1 0	2 1 3 1 0	
P3	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of x for which this is a safe state based on the banker's algorithm?

X=0

S. Compare the circular wait scheme with the deadlock avoidance schemes concerning the following issues: run time overhead system throughput

Circular Wait Scheme:

Run-time Overhead: The circular wait scheme is a deadlock prevention strategy that ensures that processes always request resources in a specific order. This scheme does not impose significant run-time overhead since it does not involve complex resource allocation and tracking algorithms. The overhead mainly depends on the order in which resources are requested and released.

System Throughput: The circular wait scheme does not directly impact system throughput. However, since it enforces a specific order for resource requests, it may lead to lower concurrency and resource utilization. If multiple processes are waiting for resources in a circular manner, it can result in resource contention and reduced overall system throughput.

Deadlock Avoidance Schemes:

Run-time Overhead: Deadlock avoidance schemes involve more complex algorithms for resource allocation and deadlock detection. These schemes require additional bookkeeping and monitoring of resource allocation and request patterns. The run-time overhead associated with deadlock avoidance schemes is typically higher than the circular wait scheme. The algorithms used for deadlock avoidance may involve resource allocation graphs, banker's algorithms, or other dynamic algorithms.

System Throughput: Deadlock avoidance schemes aim to maximize system throughput by dynamically allocating resources to processes while ensuring that the system remains in a safe state. By carefully analyzing resource allocation requests and resource availability, these schemes

can effectively avoid deadlocks and allow processes to execute concurrently. However, the system throughput may still be affected if the resource allocation and scheduling algorithms are not optimized or if resource contention occurs.

In summary, the circular wait scheme has a lower run-time overhead compared to deadlock avoidance schemes but may lead to lower system throughput due to resource contention. Deadlock avoidance schemes have higher run-time overhead but aim to maximize system throughput by dynamically managing resource allocation to prevent deadlocks. The choice of the scheme depends on the specific requirements and characteristics of the system.

T . Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB, and 250 KB. In that order, these partitions need to be allocated to four processes of sizes 357 KB, 210 KB, 468 KB, and 491 KB.

Perform the allocation of processes using

i First Fit Algorithm

200 KB	400KB	600KB	500KB	300KB	250KB
	357KB	210KB	468KB		

The process of 491KB should in waiting

ii Best Fit Algorithm

200 KB	400KB	600KB	500KB	300KB	250KB
	357KB	491KB	468KB		210KB

iii Worst Fit Algorithm

200 KB	400KB	600KB	500KB	300KB	250KB
		357KB	210KB		

Processes of 468KB and 491KB should in waiting

U. What is the difference between logic and global page allocation? What are their respective advantages and disadvantages?

Logic (or Virtual) Page Allocation:

Logic page allocation is a technique used in virtual memory systems to manage the mapping between virtual addresses used by processes and physical addresses in the physical memory (RAM).

Each process has its own logical address space, which is divided into fixed-size units called logical pages.

These logical pages are mapped to physical pages in the physical memory using a data structure called a page table.

The operating system is responsible for managing the page table and translating logical addresses to physical addresses during memory access.

Advantages of logic page allocation:

It allows processes to have a larger logical address space than the available physical memory, enabling efficient utilization of memory resources.

It provides memory isolation between processes, preventing one process from accessing or modifying the memory of another process.

Disadvantages of logic page allocation:

It introduces overhead due to the need for address translation, which adds latency to memory accesses.

It requires additional memory to store the page table, which can consume significant memory resources in systems with a large number of processes.

Global Page Allocation:

Global page allocation is a memory management technique that assigns physical memory pages to all processes in the system at the time of process creation or system boot.

In global page allocation, each process receives a fixed set of physical pages allocated to it throughout its lifetime.

Processes directly access physical memory without the need for address translation.

Advantages of global page allocation:

It provides faster memory access since there is no need for address translation.

It reduces the memory overhead associated with maintaining page tables.

Disadvantages of global page allocation:

It limits the maximum number of processes that can run concurrently to the available physical memory capacity.

It may result in inefficient memory utilization if processes do not fully utilize their allocated physical pages.

In summary, logic page allocation provides virtual memory management and allows efficient memory utilization and memory isolation but introduces overhead due to address translation. On the other hand, global page allocation provides faster memory access and reduces memory overhead but limits the number of concurrent processes and may lead to inefficient memory utilization. The choice between these techniques depends on the specific requirements and constraints of the system.

V. Calculate the size of memory if its address consists of 22 bits and the memory is 2-byte addressable.

Number of addresses = 2^(number of bits)
= 2^22 = 4,194,304

Memory size = Number of addresses * 2 bytes
= 4,194,304 * 2 bytes
= 8,388,608 bytes
= 8.3 MB ≈ 8 MB

W. Calculate the number of bits required in the address for a memory having a size of 16 GB. Assume the memory is 4-byte addressable.

Let 'n' number of bits are required. Then, the Size of memory = 2^n x 4 bytes.

Since the given memory has a size of 16 GB, so we have-

2^n x 4 bytes = 16 GB

2^n x 4 = 16 G

2^n x 2^2 = 2^{34}

2^n = 2^{32}

∴ n = 32 bits

X. Consider a machine with 64 MB of physical memory and a 32-bit virtual address space. If the page size is 4 KB, what is the approximate size of the page table?

- Size of main memory = 64 MB

- Number of bits in virtual address space = 32 bits
- Page size = 4 KB

Size of main memory

= 64 MB

= 2^{26} B

Thus, the number of bits in physical address = 26 bits

Number of frames in main memory

= Size of main memory / Frame size

= 64 MB / 4 KB

= 2^{26} B / 2^{12} B

= 2^{14}

Thus, Number of bits in frame number = 14 bits

We have,

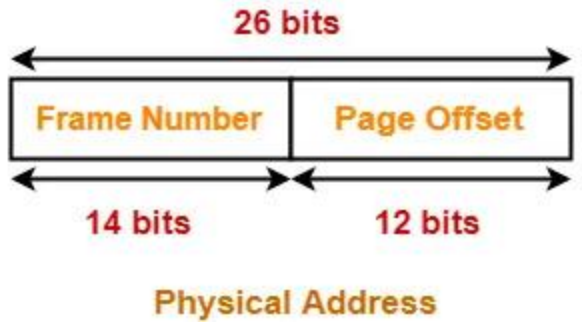
Page size

= 4 KB

= 2^{12} B

Thus, Number of bits in page offset = 12 bits

So, Physical address is-



Number of bits in virtual address space = 32 bits

Thus,

Process size

= 2^{32} B

= 4 GB

Number of pages the process is divided

= Process size / Page size

= 4 GB / 4 KB

= 2^{20} pages

Thus, Number of entries in page table = 2^{20} entries

Page table size

= Number of entries in page table x Page table entry size

= Number of entries in page table x Number of bits in frame number

= 2^{20} x 14 bits

= 2^{20} x 16 bits (Approximating 14 bits ≈ 16 bits)

= 2^{20} x 2 bytes

= 2 MB

Consider a system with byte-addressable memory, 32-bit logical address, 4 KB page size, and page table entries of 4 bytes each. Calculate the size of the page table in the system.

Given information:

- Byte-addressable memory
- 32-bit logical addresses
- 4 kilobyte page size
- Page table entries of 4 bytes each

Calculating the number of pages:

- 32-bit logical addresses can address 2^{32} bytes of memory
- Dividing this by the page size of 4 kilobytes (or 2^{12} bytes) gives 2^{20} pages

Calculating the size of the page table:

- Each page table entry is 4 bytes
- Multiplying this by the number of pages gives $4 * 2^{20}$ bytes
- Converting this to megabytes gives 4 MB

Therefore, the size of the page table in the system is 4 megabytes.

Consider a single-level paging scheme. The virtual address space is 4 MB and the page size is 4KB. What is the maximum page table entry size possible such that the entire page table fits well on one page?

Number of Pages of Process-

Number of pages the process is divided= Process size / Page size= 4 MB / 4 KB= 2^{10}pages

Page Table Size-

Let page table entry size = B bytes

Now,

Page table size

= Number of entries in the page table x Page table entry size

= Number of pages the process is divided x Page table entry size

= 2^{10}x B bytes

Now,

According to the above condition, we must have-

2^{10}x B bytes <= 4 KB

2^{10}x B <= 212

B <= 4

Thus, the maximum page table entry size possible = 4 bytes

Consider a single-level paging scheme. The virtual address space is 16 GB and the page table entry size is 4 bytes. What is the minimum size possible such that the entire page table fits well on one page?

Virtual address space: 16 GB

Page table entry size: 4 bytes

To calculate the minimum size required for the page table, we need to determine the number of entries in the page table. Each entry in the page table corresponds to a page in the virtual address space.

Since the virtual address space is 16 GB, and the page size is not mentioned, let's assume a commonly used page size of 4 KB (4096 bytes).

Number of pages in the virtual address space = Virtual address space / Page size

Number of pages = (16 GB) / (4 KB) = (2^34 bytes) / (2^12 bytes) = 2^(34-12) = 2^22 pages

Now, to fit all these pages in the page table, we need to determine the number of entries required. Each entry in the page table corresponds to one page, so the number of entries is equal to the number of pages.

Number of page table entries = Number of pages = 2^22

Since each page table entry is 4 bytes in size, the minimum size required for the page table to fit on one page is:

Page table size = Number of page table entries * Size of each entry

Page table size = (2^22) * 4 bytes = 2^(22+2) bytes = 2^24 bytes

Therefore, the minimum size required for the page table to fit on one page is 16 MB (2^24 bytes).

Y. explain 2 most popular operating systems for the internet of things

1. FreeRTOS:

- FreeRTOS (Real-Time Operating System) is a lightweight, open-source operating system specifically designed for resource-constrained IoT devices.
- It offers real-time scheduling capabilities and efficient task management, making it suitable for IoT applications with strict timing requirements.
- FreeRTOS provides a small footprint, low latency, and low power consumption, which are crucial for IoT devices with limited resources.
- It supports a wide range of microcontrollers and architectures, allowing flexibility in hardware choices.
- FreeRTOS has a rich ecosystem, providing a variety of libraries, device drivers, and tools to aid in IoT development.
- It offers robust task scheduling, inter-task communication, and synchronization mechanisms, ensuring reliable and secure operation.
- FreeRTOS is widely adopted and has a large community of developers, which contributes to its continuous improvement and support.

2. Linux-based operating systems (e.g., Debian, Ubuntu, Raspbian):

- Linux-based operating systems are prevalent in the IoT space, leveraging the power and versatility of the Linux kernel.
- These operating systems are derived from popular Linux distributions, such as Debian or Ubuntu, and adapted to run on IoT devices.
- Linux-based OSes provide a wide range of software packages and libraries, enabling easy development and customization of IoT applications.
- They offer strong networking capabilities, allowing seamless integration with internet protocols and services.
- Linux-based systems benefit from the extensive Linux community, which provides continuous support, security updates, and development resources.

- These operating systems can run on a variety of hardware platforms, ranging from small single-board computers (e.g., Raspberry Pi) to more powerful IoT gateways and servers.
- Linux-based OSes offer robust security features and access controls, crucial for protecting IoT devices and data.

Z. Discuss 2 differences between general purpose operating system and real-time operating system

Task Scheduling: GPOS uses preemptive multitasking with priority and time slice-based scheduling, while RTOS uses deterministic scheduling algorithms to meet strict timing requirements.

Resource Management: GPOS provides extensive resource management features and aims for efficient utilization, while RTOS prioritizes minimal resource usage and has a smaller footprint.

In general, an operating system (OS) is responsible for managing the hardware resources of a computer and hosting applications that run on the computer. An RTOS performs these tasks but is also specially designed to run applications with very precise timing and a high degree of reliability.

A. consider the series of releases of Windows, Linux, OS X, What drives the changes in OS? What are the most compelling issues facing OSes today

1. Technological Advancements: OSes need to keep pace with advancements in hardware, networking, and software technologies. This includes support for new processors, devices, file systems, networking protocols, and emerging software development paradigms.
2. User Needs and Expectations: OSes strive to meet the evolving needs and expectations of users. This includes improvements in user interfaces, accessibility features, application compatibility, and user experience enhancements. OSes may also adapt to changing work patterns and lifestyles, such as the growing demand for mobile computing or cloud-based services.
3. Security and Privacy: Security is a major concern for OSes, and they continuously evolve to address vulnerabilities and protect against emerging threats. OS updates often include security patches, enhancements to access controls, encryption mechanisms, and privacy features to safeguard user data.
4. Competitive Market: OS vendors face competition from each other, driving them to innovate and differentiate their offerings. Each release aims to introduce new features,

improved performance, and better compatibility to attract users and maintain a competitive edge.

5. Developer Ecosystem: OSes need to provide robust development tools, APIs, and frameworks to support application development. Supporting a vibrant and thriving developer ecosystem encourages the creation of diverse and innovative applications, which further enhances the value of the OS.

The most compelling issues facing OSes today include:

1. Security: As cyber threats continue to evolve, ensuring robust security measures and timely patching is critical. OSes face challenges in protecting against malware, data breaches, and vulnerabilities in software components.
2. Privacy and Data Protection: With the increasing amount of personal data being collected and processed, OSes need to address privacy concerns and provide users with granular control over their data.
3. Compatibility and Integration: OSes must ensure compatibility with a wide range of hardware devices, software applications, and emerging technologies. They also face challenges in providing seamless integration across multiple platforms and ecosystems.
4. Performance and Efficiency: OSes strive to optimize performance and resource utilization to deliver fast and efficient user experiences. This includes minimizing boot times, reducing memory and energy consumption, and optimizing task scheduling.
5. Accessibility and Inclusivity: OSes need to consider diverse user needs, including accessibility features for individuals with disabilities, multilingual support, and inclusive design principles.

B. explain how operating systems have evolved over the years and provide 3 reasons for this evolution

Technological advancements: OSes have adapted to new hardware technologies, such as multi-core processors and virtualization, to improve performance, scalability, and resource utilization.

User needs and expectations: OSes have undergone changes to meet user demands for user-friendly interfaces, enhanced multimedia capabilities, better application compatibility, and seamless integration with emerging technologies.

Security and privacy concerns: OSes have evolved to address vulnerabilities, protect against malware and cyber threats, and provide robust security features to safeguard user data and privacy.

1. Technological Advancements: As hardware technologies have advanced, operating systems have evolved to leverage new capabilities and accommodate changing

hardware architectures. Examples include the transition from single-core to multi-core processors, the adoption of virtualization techniques, support for graphical user interfaces (GUI), and the integration of networking capabilities. OSes have evolved to effectively manage and utilize these advancements, improving performance, scalability, and resource allocation.

2. Shifting User Needs and Expectations: User needs and expectations have driven the evolution of operating systems. As computing became more accessible and widespread, users demanded user-friendly interfaces, enhanced multimedia capabilities, better application compatibility, and improved productivity tools. The evolution of operating systems has aimed to provide intuitive interfaces, greater software support, enhanced multimedia experiences, and seamless integration with emerging technologies, such as mobile devices and cloud computing.
3. Security and Privacy Concerns: With the growth of interconnected systems and digital services, security and privacy have become critical concerns. Operating systems have evolved to address vulnerabilities, protect against malware and cyber threats, and provide robust security features. Updates include enhancements in access controls, authentication mechanisms, encryption techniques, and secure boot processes. The evolution of operating systems also incorporates privacy features to give users more control over their data and mitigate privacy risks.
4. Advancements in Distributed Computing and Connectivity: The proliferation of networked systems, the internet, and distributed computing models have driven the evolution of operating systems. OSes have adapted to facilitate seamless network connectivity, support remote access and collaboration, and enable distributed processing and data storage. This includes advancements in network protocols, file-sharing mechanisms, distributed file systems, and cloud computing integration.
5. Changing Computing Paradigms: The evolution of operating systems has been influenced by shifting computing paradigms. For instance, the rise of mobile computing led to the development of mobile operating systems tailored for smartphones and tablets, emphasizing touch interfaces, app stores, and mobile-specific features. Similarly, the growth of cloud computing resulted in operating systems optimized for virtualized environments, scalable resource management, and cloud service integration.

suggest 4 programming languages for a new modern OS built from scratch

1. C:
    - C is a low-level programming language that offers close interaction with hardware, making it a popular choice for operating system development.

- It provides direct access to memory and hardware resources, allowing for fine-grained control over system operations.
- C has a long history of being used in OS development, with many tools, libraries, and documentation available to support this purpose.
- It offers a balance between performance and control, making it suitable for low-level tasks such as memory management, device drivers, and kernel development.

2. C++:
    - C++ is an extension of the C language that provides additional features such as object-oriented programming, templates, and stronger type-checking.
    - It offers a higher level of abstraction and can help in structuring complex systems and maintaining code modularity.
    - C++ is widely used in operating system development, particularly for building kernel components and device drivers.
    - The language provides powerful libraries, such as the Standard Template Library (STL), that can aid in development efficiency.
3. Rust:
    - Rust is a relatively new systems programming language that focuses on safety, concurrency, and memory management without sacrificing performance.
    - It provides strong memory safety guarantees through its ownership and borrowing system, reducing the risk of memory-related bugs.
    - Rust offers built-in support for concurrent programming, making it suitable for developing highly scalable and efficient OS components.
    - The language's emphasis on safety and modern language design principles makes it an attractive choice for building a new OS with a focus on security and reliability.
4. Assembly Language:
    - Assembly language is a low-level programming language that provides a direct representation of machine code instructions.
    - It offers the highest level of control over hardware resources, enabling precise manipulation of system registers and instructions.
    - Assembly language is commonly used for writing critical parts of an operating system, such as bootloaders, interrupt handlers, and context-switching routines.
    - While not a primary language for developing an entire OS, knowledge of assembly language is essential for OS developers to understand system internals and optimize critical sections.

Suppose you are hired to choose “the best operating system”. How would you decide? What do you look for? Consider in following

Briefly outline your strategy

List 5 metrics you would use and their relative weights

Strategy Outline:

Identify Requirements: Understand the specific needs and requirements of the organization or project for which the operating system will be used. Consider factors such as performance, scalability, security, hardware support, software compatibility, ease of use, and development ecosystem.

Research Available Options: Conduct thorough research on existing operating systems to gather information about their features, capabilities, community support, track record, and market adoption. Consider both commercial and open-source offerings.

Evaluate Metrics: Determine a set of metrics that align with the identified requirements and rank the operating systems based on their performance in these metrics.

Weighted Evaluation: Assign relative weights to the metrics based on their importance in meeting the organization's needs. This weighting helps prioritize the metrics and allows for a more objective evaluation.

Assess Trade-offs: Consider the trade-offs associated with each operating system. Some metrics may have conflicting requirements, so it's important to understand the implications and identify the system that best aligns with the organization's priorities.

Pilot Testing and Feedback: Consider conducting pilot tests or obtaining feedback from relevant stakeholders, such as IT professionals or end-users, to gather practical insights and validate the evaluation results.

Final Decision: Based on the evaluation, the relative weights assigned to the metrics, and the feedback obtained, make an informed decision on the best operating system that meets the organization's requirements.

Metrics and Relative Weights:

Performance (30%): Evaluate the operating system's efficiency, responsiveness, and throughput in handling various workloads. Consider factors such as CPU utilization, memory management, disk I/O, and network performance.

Security (25%): Assess the security measures and features offered by the operating system, including access control, authentication mechanisms, encryption support, vulnerability patching, and audit capabilities.

Software Ecosystem (20%): Evaluate the availability and quality of software applications, libraries, and development tools compatible with the operating system. Consider factors such as the size of the software repository, community support, and developer-friendly features.

Stability and Reliability (15%): Examine the track record of the operating system in terms of stability, reliability, and bug-fixing. Look for historical data on system crashes, known vulnerabilities, and the frequency of updates and patches.

User-Friendliness (10%): Consider the ease of use and user experience the operating system provides. Evaluate factors such as the intuitiveness of the user interface, availability of documentation and support resources, and the learning curve required for administrators and end-users.

Dhimantha T.S

Faculty of Technology