Journal of
Operating Systems
Development & Trends

ISSN: 2454-9355
Volume 13, Issue 1, 2026
January–April
DOI (Journal): 10.37591/JoOSDT

https://journals.stmjournals.com/joosdt

Review JoOSDT

The Interface of Hardware and Intelligence: The Function of Operating Systems

V. Basil Hans*

Abstract

Operating systems play a central role in bridging the gap between computer hardware and user interaction. They simplify complex machine-level operations and transform them into user-friendly and efficient digital experiences. At their core, operating systems are responsible for managing essential tasks such as process scheduling, memory allocation, file system organization, and device coordination. By handling these functions effectively, they ensure that hardware resources are used in an optimal and balanced way. In addition to resource management, operating systems contribute significantly to system security and stability. They protect data, prevent unauthorized access, and reduce the chances of system crashes, allowing users to perform both basic and advanced computing tasks with confidence. Without an operating system, interacting with hardware would be extremely difficult and time-consuming. Over time, operating systems have evolved remarkably. Early systems relied heavily on command-line interfaces, requiring technical expertise, while modern operating systems offer intuitive graphical interfaces and intelligent features. Today's platforms are more adaptive, supporting multi-tasking, cloud integration, and mobile computing. This continuous evolution highlights how operating systems remain a crucial intersection between hardware capabilities and human needs in an ever-changing technological landscape.

Keywords: Drivers for devices, file system, hardware for computers, managing memory, managing software, operating systems (OS), scheduling processes, user interface

INTRODUCTION

All physical systems have limited resources that must be shared in an orderly manner [1]. Bandwidth, computing power, and storage space are some of the resources available to a computer. A hardware-software barrier splits these resources into two groups: hardware and software. A robust interface must be established so that hardware and software can communicate with each other on how to share resources. An operating system (OS) oversees the establishment of this connection. More generally, it is a set of principles that allows hardware and software to work together in a useful way. Many complex factors make this interface difficult to develop. Hardware resources are organized in a hierarchy, and software can simultaneously interact with hardware in several ways at the same time. After the bootup process, the application software must regain control of the hardware without breaking the confidence given to the OS.

The main jobs of an OS are to show and negotiate resources, provide a virtualized picture of hardware, and set rules and progress guarantees for how these resources can be used and shared. The resource negotiation process begins as soon as the hardware is turned on. System administrators choose the allocation of resources for OS processes; this first negotiation is facilitated by an additional software

***Author for Correspondence**
V. Basil Hans
E-mail: vhans2011@gmail.com

[1]Research Professor, Department of Management and Commerce, Srinivas University, Mangaluru, Karnataka, India.

Received Date: March 27, 2026
Accepted Date: April 04, 2026
Published Date: April 30, 2026

Citation: V. Basil Hans. The Interface of Hardware and Intelligence: The Function of Operating Systems. Journal of Operating Systems Development & Trends. 2026;13(1):7–20p.

component. All other types of resources must be placed in a hierarchy that shows how they work and how they are built into the hardware. In a multilayer architecture, adjacent components may directly negotiate resources or provide complementary abstractions that streamline the discussions among the processing units affixed to these components.

Research Objectives

The main objectives of this study are:

- To explain how OSs act as an interface between hardware and software.
- To analyze key OS functions, such as process management, memory handling, and I/O coordination.
- To examine modern trends like virtualization, containers, and heterogeneous computing.
- To evaluate performance, security, and reliability aspects in OSs.

Research Contribution

This paper provides:

- A structured explanation* of OS as an interface between hardware and intelligence.
- A comparative understanding* of traditional and modern OS techniques.
- Inclusion of practical examples and case-based explanations*.
- Identification of research gaps in scheduling, heterogeneous systems, and OS design*.

METHODOLOGY

This study followed a *qualitative and analytical research method.* This is based on a review and comparison of existing literature, OS designs, and real-world implementations.

The methodology includes:

- *Literature analysis:* Study of previous research papers, technical reports, and OS documentation.
- *Comparative study:* Comparison of different OS techniques, such as scheduling algorithms, memory management, and virtualization.
- *Case-based understanding:* Examples from real systems, such as Linux, Windows, and container-based environments.
- *Conceptual modeling:* Explanation of OS structure using layered and modular approaches.

Example Case Study

A simple case is the comparison between:

- *Traditional virtual machines (VMs):* High isolation but more overhead.
- *Containers:* Lightweight and faster but share the host OS kernel.

This comparison shows how modern OS design improves efficiency while balancing security.

LITERATURE REVIEW AND RESEARCH GAP

Previous studies have discussed OSs mainly as *resource managers and control programs*. For example:

- Early research focused on batch processing systems and efficiency.
- Later studies introduced multi-programming and scheduling algorithms.
- Recent work explores virtualization, cloud computing, and containerization.

However, there are key limitations:

- Many studies focus only on individual components like scheduling or memory.
- Limited research connects hardware evolution with OS intelligence.
- Insufficient work on heterogeneous hardware management and unified OS design.
- Few studies provide practical comparisons between traditional and modern OS approaches.

Research Gap

This paper addresses:

- The lack of an integrated view of OS as a hardware–intelligence interface.
- The need for comparative and application-based analysis.
- The requirement for simplified explanations of complex OS concepts.

BASIC PRINCIPLES OF HARDWARE AND SOFTWARE INTERACTION

Hardware is the physical mechanism that performs data operations, whereas software is a set of instructions that tells the hardware what to do. When asking how hardware and software may work together, it is best to find the hardware-software border, which is the parameter that shows how they connect. Then, explain the barrier in both abstract and concrete terms. Most hardware cannot run machine instructions on its own; therefore, it must communicate with other hardware, usually control hardware, over the boundary to set up a set of instructions. A computer system has a built-in boot sequence that ensures that the steps required to reach a legitimate operational state are performed in the correct order. When an application joins the system in a non-operational state, the initialization program takes over. It does this by running a set of commands that give it authority over memory and other resources from the hardware in the system. To run an operating system (OS), a computer requires certain hardware and software capabilities. Not all devices can load an OS kernel into memory and execute it.

Even when the control is transferred to an OS kernel, many OS functions cannot be accessed until initialization is performed. A simple kernel can perform just enough to run a file system device that has a user program. Once the kernel loads the user software from the disk into memory, it can continue running beneath the program. At this time, the idea that the user application is running on its own has already become a useful feature of the OS. The kernel can use other OS-compatible devices to add more services without stopping any of the running user programs. When an OS is started up to the point where it can hand control over to an OS user application, it is still in the first stage of operation. However, the OS-capable boundaries of hardware and software have already moved forward considerably [2].

The kernel offers three key services for negotiating between the hardware and OS: it can virtualize the hardware configuration, provide a wide range of resource access abstractions, and guarantee resource access security. In a multi-user situation, system resources are often limited. A kernel that does not ensure that shared, finite resources are used fairly and sensibly cannot help users compete for those resources fairly and reasonably. Therefore, the kernel solves the challenge of controlling who can use these limited resources. The hardware resource that presents the most significant and demanding challenge to kernel management is arguably the most easily overlooked: the exclusive and unequivocal right to access any hardware resource or OS user program. For OS user software to work, it must be able to move away from the OS at all times and without any breaks. The kernel also provides access to the external memory configuration in a way that meets the needs of the user software, such as paging, segmentation, or other needs. The rise of virtual storage methods, network-style processing, and homogeneous surrogate storage addressing has greatly aided the progress of services in modern OSs [3].

Hardware Abstraction and What It Means

OS creates a vital link between software and hardware by using abstraction layers to mask the hardware specifics from the user [4]. An OS is a layer of hardware and software that sits above programmable hardware. This allows you to program with a full understanding of how the OS works without having to know how the hardware works. The OS is the initial level of abstraction. Users work at this level, not the hardware level. Therefore, the OS uses the same programming rules that were used to write the OS requirements to create an abstraction over the hardware. Most elaborations involve furnishing comprehensive specifications for an OS that functions on established programmed hardware.

The design of programmable hardware is directly influenced by the characteristics of the hardware-software interaction [5]. It is now necessary to perform hardware-software co-design at all levels, starting with the ways in which programmed hardware and the OS interact. This is because programmable devices now permit design points at the OS level. The goal of integrating hardware and software is to create an OS, which is a software description of the customized OS specified by architecture.

The system setup that works for a board prototyping program initiates the system boot. A short OS elaboration and basic hardware description were loaded into the program memory. This allows the OS to set itself up automatically and allows the hardware or software to direct the bootup sequence. Because the configuration is stored in a simple ASCII format with a limited number of operations, it is possible to add programming languages that can be read by the assembly into the initial description. The structuring and routing tools use the full and complementary compiled programming explanation of the designed hardware to create mask files that will be used in real life.

The Operating System's Part in Managing Resources

OSs serve as intermediates that negotiate the use of hardware resources on behalf of programs, manage their simultaneous execution, and offer virtualized representations of hardware resources to enhance portability and programmability across diverse conditions [5]. Different OSs support different types of hardware, but they usually have three main jobs that the OS kernel takes over from application-level software: negotiating the exclusive use of hardware resources between competing applications, virtualizing hardware resources to create a system-wide coherent view, and ensuring that access to hardware resources is consistent with the logical view set by the hardware-software interface and application programming interface [6].

OSs started as resource managers that used batch processing to get the most work done, but they did not care about response time or interaction time. The types of controlled resources have changed over the years, but the basic functions of resource management are still important for interactive apps today. Batch workloads changed to multi-programming, which made it necessary for processes to share processors, memory, and I/O devices in a way that maintained a high throughput [7]. Sadly, the best-effort, priority-driven policies that came out of batch processing remained the same when multi-programming changed to multi-tasking and interactive job sharing. Scheduling algorithms made it easier for job phases to overlap, but they did not guarantee full responsiveness of the system. Resource managers, who did not realize how interactive they were, continued to cut costs by compromising responsiveness.

Managing Processes and Scheduling

Scheduling is one of the most important tasks of a general-purpose OS. Every application that is running requires the OS to access the processor; therefore, scheduling affects the performance of applications, user satisfaction, and the overall use of the system. The goals of modern OSs are closely related to this important service, as well as the utilization of multi-core processors, multi-process applications, cluster-based architectures, and mobile devices [8]. These tendencies make the scheduler work more efficiently. It must also meet complicated requirements because of the different loads and uses [9].

The basic unit of execution of an OS is a process. For regular workloads, such as desktop applications, web servers, and communication, support for concurrent activity has become a necessity. Therefore, an OS needs to have advanced tools to handle simple interactions between processes.

As workloads have changed, architectural support for improving context switching has also changed. Different register groups work with different processes. Starting and stopping programs usually sends state information through the cache memory instead of the main memory. Therefore, while bandwidth is still quite important, reducing latency is becoming increasingly important. The process configuration files include a lot of information about the state of the process when it has to be paused. Users can choose to pick up where they left off with stopped operations when the current workload is completed.

RESULTS / PERFORMANCE COMPARISON

Performance Comparison Example (Added Content)

A comparison of common scheduling algorithms is shown in Table 1.

Observation

- No single algorithm is best for all cases.
- Modern systems use hybrid scheduling techniques.
- Performance depends on workload and system goals.

Scheduling Algorithms and Performance Factors

One of the most important functions of an OS as a middleman between hardware and software is to provide processes access to resources. A strong scheduling algorithm is required to distribute resources fairly, and many solutions have been suggested, each of which works well for a different situation. The political consequences of resource allocation have resulted in the creation of unconventional “customer appeasement” algorithms that prioritize or even preempt allocation to processes based on user requests provided by machines or humans. Systems that change the scheduling algorithm and group processes depending on their recent history or “temporal ecology” [8] are better than considering all processes the same. We examine each of them using a set of performance metrics that measure throughput, latency, reaction time, and the extent to which resources are shared. A completely new way to schedule calls for complete separation between dedicated global, per-resource global, and per-process local scheduling is proposed. Each of these uses a different, constantly acquired “spare” resource. This method creates a single global schedule and uses multi-objective, competitive tuning algorithms across resource arrays and proportions, which is different from most other methods [10].

Consistency, Concurrency, and Synchronization

Multiple processes can run and talk to each other at the same time when they share a single physical memory system. This allows different programs to run, sometimes simultaneously and sometimes at different times. Running programs in parallel improves both the throughput and latency. Keeping resources free for real-time activities also helps ensure that things happen on time. To balance these goals, OS process management uses several methods. Hardware-oriented abstractions are required to schedule processes consistently.

When the time between the execution of the two processes does not overlap, they run simultaneously. Concurrent execution includes numerous processes and several threads that are part of the same process. A synchronization method is needed to ensure that multiple users can use shared resources simultaneously without changing the resource consistency [11]. Process scheduling provides the resources that are needed to start a process, such as the Central Processing Unit (CPU), address space, and I/O channels. This implies that the scheduling method sets the concurrency mechanism. Concurrency should include multiplexing, overlapping, and preempting. Scheduling allows some processes to perform computations for a short time, which gives others the chance to start and finish execution.

There are two types of concurrency: cooperative and preemptive concurrency. Modern OSs support both these approaches. In cooperative systems, a process runs until it is completed, gives up resources on its own, or makes a blocking call (such as I/O in an OS). In preemptive systems, the kernel can stop running and provide control over another process at any time.

Table 1. CPU scheduling comparison.

Algorithm	Response time	Throughput	Complexity	Use case
FCFS	High	Low	Simple	Batch systems
SJF	Low	High	Moderate	Short jobs
Round Robin	Moderate	Moderate	Simple	Time-sharing systems
Priority Scheduling	Variable	High	Complex	Real-time systems

Some OSs allow limited preemption, which means that control must be given up during certain kernel services or between primitive instructions when the user runs the program. In some cases, preemption limits the set of system primitives that can be used.

MEMORY HIERARCHY AND ADDRESS TRANSLATION

Modern computers use a memory hierarchy to achieve the right balance between performance, energy, and cost. The memory hierarchy is a list of different forms of storage arranged by speed and size. The fastest and smallest memories are at the top, and the slowest and largest memories are at the bottom of the table. You can access faster storage types faster and with more bandwidth than slower types. For example, throughput, energy, and cost are all rates, and the units usually contain one or more bytes divided by a time unit, which means that the amounts of data and time are divided. Various types of storage in the hierarchy have varied performance traits, and their specifications help programmers use them well. The data at the top of the hierarchy is cached, which has the largest effect on the performance of the memory system at lower levels. Virtual-to-physical address translation helps fill this gap by providing software layers with more freedom in their design and ensuring that different caches work together efficiently at the software level [12].

The computer retrieves data from secondary storage if available. If not, the Memory Management Unit (MMU) generates a page fault and moves to the page fault management routine. The MMU uses address translation to convert virtual addresses created by user programs into physical addresses that the memory unit can utilize. To save this extra work, a translation lookaside buffer (TLB) is sometimes employed to store the most recently used virtual-to-physical address mappings. Process control blocks (PCBs) hold the address base and limit variables that are linked to the process. This keeps the address space separate. Address space isolation ensures that one user application cannot change how another user program runs, which ensures that programs can run simultaneously [13].

Practical Example of Memory Management

In modern systems:

- When a user opens multiple applications, the OS uses virtual memory.
- If RAM is full, inactive pages are moved to disk (paging).
- This allows smooth multi-tasking without a system crash.

This demonstrates how OS improves efficiency and user experience.

Strategies for Virtual Memory and Paging

Virtual memory solves the problems of directly addressed memory by allowing secondary storage to keep bits of memory that are not being utilized. This frees up more memory for the applications. With the paging approach, parts of the memory can be moved to the disk, making the RAM available to apps without them having to do anything. Virtual memory also keeps processes separate by preventing them from using memory that is not assigned to their context. This makes the system more secure and reliable [13].

Virtual memory technology has significantly affected the evolution of computer systems, and it depends on CPU support, such as an MMU. Paging is a common mechanism that splits the system into memory resource providers and consumers, with the CPU acting as a middleman using a present flag and page fault exception. Because of these qualities, memory management can occur silently and dynamically, which means that the memory manager can provide or take back memory without the consumer having to do anything. When there are multiple memory managers, each one must manage its own resources. A hierarchical management chain is typically too complicated and does not work with the original concept of virtual memory. The memory manager and consumer mostly communicate with each other through page faults and CPU protocols. The memory manager is trusted to provide resources and manage the address space. The memory manager's job includes handling page faults to fulfill resource requests and continue execution. Virtual memory allows swappable and unswappable memory, memory-mapped files, device access, security, process separation, shared memory, and dynamic link library (DLL) management. This is especially true for Windows and Linux systems [14].

When RAM was hard to come by, Unix virtual memory was built, and demand paging was necessary for the system to function. Paging completely separates the virtual memory from the backing storage such that each process sees the same virtual address space. Copy-on-write is the most fundamental virtual memory primitive, and page coloring can be used to take advantage of this. Workloads increasingly focus on memory bandwidth and locality, and they often pin pages to ensure that they work and are correct. Modern MMUs allow mappings at a coarser level than individual pages, including superpages, and can handle massive and gigantic pages through kernel configurations and dynamic resizing pools [15].

Memory Safety and Cache Coherence

Systems generally run sequential programs, yet processors can only run a few instructions at a time. Data hazards happen when a program reads and modifies the same data at the same time, using old values [16]. Instructions that have read-after-write dependencies are another source of risks. Another one is writing the same data at the same time, but in a way that is not always the same. Modern OSs can handle checkpoints based only on address traces and control flow. The idea of safety is that the machine has to be able to switch between a store that keeps overwriting l and a store that writes to a read from a l. Many-core CPUs and heterogeneous systems have cache coherence protocols built in. These protocols let separate caches store copies of the same data at the same time, but only for a short time [17]. In a multi-core architecture, performance deteriorates during the intensified scenario application. A memory controller on the chip microprocessor accesses the cache, but there is no way to guarantee that the typical cache-coherent technique will work under robust assurance. The commonly used prediction application or service in cloud environments might, therefore, serve as a reference. Cache coherence is a method that lets you add separate, unrelated threads to the same platform without changing the basic programming principles of an application. Instruction skipping or out-of-order execution are ways to improve performance without losing overall parallelism or adding more programming work. The main application service and the arrival data share the rich device references and initial application settings,, which makes them quite short even when the scenario is aware. In addition, an accelerator might actively participate without defining a programming job at the same time.

INTEGRATION OF I/O SYSTEMS AND PERIPHERALS

OSs provide I/O devices with the tools they require to function. Each peripheral is linked to a device driver. It creates a device-specific interface and hides raw communication semantics. Drivers provide high-level functionality and connect to the OS kernel to enable low-latency operations and fast data transmission. Device drivers can also be organized in layers to make them more modular. For example, a storage device can have a logical volume manager on top of the file system, which is built on top of a block storage driver that is specific to that device.

The kernel communicates with devices using standard open-source APIs and interfaces for the device type. This allows user-level programs to access devices directly or indirectly (via a hypervisor). It also works with devices that run asynchronously by allowing them to make interruptions whenever they need to be serviced. When an interruption occurs, the kernel calls a specific interrupt handler in the device driver linked to it.

As the gap between the processor and I/O device performance decreases, the cost of moving data inside the computer becomes a more significant problem. Currently, only a handful of modern I/O execution paths use mass storage devices that operate over hundreds of microseconds or longer. Storage, file systems, and networking implementations are no longer just about having a fast data transfer buffer. In the past, data or byte-level transfers might have disguised a lot of costs, but now the main problems are excessive context switching and poorly optimized data layouts. Researchers are continually looking for ways to improve the relative I/O efficiency. Some of these include hierarchical I/O schedulers that can handle different types of devices, I/O parallelism that can speed up simple file operations, and distributed processing across several storage nodes.

Kernel Interfaces and Device Drivers

I/O systems create a link between the hardware-software and user program boundaries [18]. These systems link external peripherals to OSs, providing users with the I/O operations they require and liberating machines from the unique features of each I/O device. Device drivers meet the basic operational needs of peripheral devices that handle data before they can communicate with OSs and user rights. Kernel interfaces provide users access to certain I/O functions, and OSs control the order in which I/O instructions are called.

OSs combine I/O subsystems and provide programmers with ways to perform input and output operations. The choice of I/O discipline is important because it has a significant impact on the performance of the entire system, including the order in which instructions are executed. The system performance generally depends on the I/O characteristics because I/O storage devices are many times slower than the CPU. Therefore, coordinated resource management and good scheduling practices are important for ensuring that systems run as efficiently as possible.

Efficiency of Storage, Networking, and Peripherals

Many computer resources are used for storage, networking, and peripheral devices; however, their efficiency remains a hotly discussed issue. Different tactics and features of OSs affect this efficiency.

The rapid growth of storage, network bandwidth, and peripheral performance and intelligence has made I/O capabilities much more than what the average desktop user requires [19]. Improving this efficiency, especially in places with "quiet" workstations and powerful PCs, is a significant challenge. For example, workstations do not usually require more than a few kilobytes per second of service via Ethernet; however, system configurations still attempt to obtain tens of megabits per second of service. Over time, bottlenecks built up as neighboring systems improved significantly. Local disks with transfer rates of 100 megabytes per minute and high-performance page buffers that access 50 megabytes or more per minute are now common in workstations. However, Ethernet controllers often share the same base memory region address space as disks and still use the runtime system page space for transferred data words [20]. The requesting-client procedure adds another layer of difficulty because it often works much lower than the maximum for the consumer device.

SAFETY, PRIVACY, AND RELIABILITY

Operating systems (OSes) are important parts of all system software stacks. They operate as a bridge between hardware and applications. One of the main jobs of an OS is resource management. This includes setting clear boundaries between the OS and users and making sure that multiple applications can share resources offered by hardware platforms in a safe and efficient way. All hardware resources, like storage, networking equipment, and peripherals, have OS level application interfaces. This means that programs do not have to worry about hardware details, and access can be controlled at a higher level [21]. When multiple applications are using hardware resources at the same time, they often share many control points, such as registers and caches. OSs set, enforce, and maintain isolation guarantees to make sure that one application accessing a resource does not affect other applications that are also using that resource.

Protection methods and their related policy models, such as access control lists (ACLs) and capabilities, ensure that both the OS code and user applications can access hardware resources. OS level access control is still an important method for protecting the use of hardware resources. OS level implementations not only provide access control but also guarantee isolation by defining the sets of system calls, memory regions, and other dependencies required by software entities. Even though operating systems (OSes) keep hardware resources separate between OS-code-level and application-level accesses, components, and code that are installed at the OS level may create extra dependencies on applications that run in the user space.

Threat Models in Operating Systems

OSs are necessary for keeping computer platforms safe and are important for building trust in the system. Therefore, security features such as confidentiality, integrity, and availability are very important aspects of the entire design of an OS. Numerous organization- and user-level security models require rules to be followed at the OS level simultaneously. This is because most computing platforms are still systems that may be used by numerous companies and organizations, which makes it easy for malware and hacking to spread. Therefore, we can develop several threat models to consider the possible weaknesses of OSs, the level of coverage needed to ensure that people have a lot of trust in the overall secure setup of a computer system, and the possible ways to reduce those risks.

In an investigation [22], the OS threat model suggests six key assumptions about the OS that an attacker could use if they were removed. These assumptions discuss the importance of physical access to compromise the computing environment, the need to define security roles for users and ensure that they follow them, the need to keep OS patches up to date for established security coverage, the need to protect the integrity of the overall system on secure start-up, the need for timely updates to stay aware of recent vulnerabilities, and the tools and methods for dealing with proof of integrity on third-party code. As the industry evolves toward more automation of installing and running software from many different sources with little or no meaningful human oversight or evaluation, each of the six remains important.

Ways to Protect and Control Access

An OS manages the resources of a computer and ensures that programs run in the right order so that user requests can be met even when hardware is limited. Some of the most well-known responsibilities of an OS are loading programs, scheduling processes, maximizing memory usage, and providing device drivers for external hardware. In addition, in keeping with a layered approach to abstraction, OSs allow application programs to run without having to know a lot about the hardware components and how they work. OSs strengthen this abstraction by creating a way for a program to run smoothly without being interrupted by other processes. However, OS concepts have considerably wider effects than just managing running apps [23].

The basic concepts of OS design are based on two main goals: protecting one program from interference or data corruption by other programs and enabling numerous applications to run simultaneously without affecting each other. The OS manages these tasks by creating partitions or protection domains that keep their memory areas separate. OSs provide two main ways to partition: ACLs and capabilities, which are based on the usual naming conventions of entities in information systems [24].

VIRTUALIZATION, CONTAINERS, AND THE GROWTH OF ABSTRACTION

Virtualization has recently gained popularity because of its benefits and the fact that modern processors have optimized virtualization instructions that make it possible to build safe, separate, and energy-efficient environments. Cloud computing is a new way of doing business that combines the best parts of Grid Computing and Virtualization. It allows many people to share resources in a way that is always available, can be scaled up or down, and only costs money when needed. However, high-performance computing (HPC) applications do not function well on cloud systems. This is primarily because hypervisors add virtualization costs, making it difficult to obtain processor instructions that are specifically designed to improve speed. Containers circumvent this problem by not using the hypervisor and instead using kernel capabilities, such as namespaces and control groups (cgroups), to offer isolation and resource accounting. This accelerates the development, distribution, and deployment of applications. Containers offer the necessary mobility and repeatability for scientific computing without the extra work associated with hypervisor-based systems. Singularity is an example of this, as it allows computational science to move to the cloud.

Modern container-based virtualization environments allow you to pack mission-critical applications into containers that contain all the libraries and executables required to run them. This is a quick and standardized method. Consequently, the encapsulated applications can run smoothly on many platforms, such as workstations, clusters, and clouds, with the required input and output. Additionally, containers can be employed for HPC workloads in specific scenarios, such as rapid prototyping from development to production environments and repeatable research. These unique characteristics can help manage data-centric applications along the computation workflow and keep applications reproducible in big-data environments or massive scientific codes in computational research, which are becoming increasingly important challenges.

Hypervisors and Resource Encapsulation

Virtualization uses two methods to create resource encapsulation in a shared infrastructure: resource virtualization and hardware isolation. Hypervisors implement these methodologies at the resource management virtualization layer [25].

Hypervisors use a VMs interface to wrap processor resources. They also enable wrapping up memory, storage, and network resources. To provide a VMs, the hosting machine must examine whether parts of different resources can be made available without compromising safety. These checks are a critical security measure that is easy to set up using representation virtualization.

Containerization vs. Full Virtualization

When applications run in a VM, they can fully encapsulate the material resources, system calls, and OS semantics. These methods have several benefits, such as better isolation (one VM cannot directly interact with another); however, using more than one VM on the same hardware makes it harder to plan for overhead and capacity. Containers are a new way to package and run workloads on an OS [26]. Containerized applications use the same kernel as the host system, which means that they have access to a larger set of global states. This means less overhead but tougher interference with limitations. However, with managed workloads, the pros can easily outweigh the cons. For instance, Kubernetes container orchestration shows how multi-programming works among managed batch-based processes. This has boosted acceptance in cloud environments [27].

The many VMs, containers, and executable apps show that there is a need for flexible abstraction over hardware resources at the boundary between the OS and applications [28]. This type of abstraction makes it possible to divide things physically, logically, and over time. It also allows a wide range of objective purposes, from sharing resources to reserving them for future use.

THINGS TO THINK ABOUT: PERFORMANCE, RELIABILITY, AND ENERGY

Performance, reliability, and energy efficiency are crucial for the OS functions mentioned above. There are several ways to evaluate performance, including throughput, latency, and resource usage. The most essential thing is to ensure that the metrics are somewhat representative of the quality of service that users perceive or some other goal for the system [29]. Reliability is not having errors while the system is running, finding them, and fixing them [30]. The purpose of reliability engineering is to accept the likelihood of problems occurring instead of trying to eliminate them. This entails developing ways to deal with them once they occur. Energy efficiency is frequently thought of as less energy, but this is not a complete definition of the term. Energy efficiency is more about how well something works per unit of energy than how much energy it uses.

Metrics and Ways to Measure

Performance measurement is important for judging the architecture of a system; therefore, it is also important for the credibility of any study that compares different systems. Only a few common metrics may be used for performance measurement, and most of them do not work for both hardware and software. For performance-oriented system software, such as OSs, distributed database systems, and network protocols, a key requirement is the correct definition and, as a result, the measurement of performance. This implies determining the amount of work that a system can handle over time [31]. This definition necessitates the differentiation of performance indicators, such as latency, throughput, and CPU utilization. Another thing to note is that the load on other parts of the system and other programs running simultaneously on the host system usually affects certain OS timing measurements, such as latency from the time a user program requests a service until it starts. This is especially true for systems where the measuring program is stored inside, makes its own input data, and provides output without the user having to do anything.

Handling Faults, Recovering from Them, and Staying Strong

In OS design, fault handling, recovery, durability, and system reliability are particularly important issues. Even in reliable systems, problems may occur that need to be addressed correctly. Consequently, methodologies for the automatic rectification of faulty data structures and self-healing systems have been investigated. Historically, fault tolerance has mostly concentrated on hardware mistakes while addressing significant software problems, particularly in OSs and applications. To fix a greater range of software failures that occur in proprietary or third-party device drivers and kernel extensions, more recovery tools are needed.

Early attempts to make system software more fault-tolerant included tandem computers and object-based OSs. Better fault isolation was achieved by protecting items that were smaller than the available protection domains and by making the protection domains more granular. Virtualization technology has made it easier to recover from disasters and keep systems running for a long time.

NEW TRENDS: DIFFERENT TYPES OF HARDWARE AND SMART SYSTEMS

The physical limits of increasing the processor clock frequency and transistor density have made power consumption and heat dissipation major issues. This has led to a shift from homogeneous to heterogeneous computing platforms that use multi-core, many-core, and supercomputing architectures. Specialized processors play a crucial role in this evolution, allowing dedicated processors to handle specific tasks that are underwhelming for general-purpose processors. The emergence of heterogeneous hardware raises three interrelated concerns: scheduling, resource allocation, and global state sharing across different cores amid diverse parallel programming paradigms within an OS [4].

Hardware Accelerators, Specialized Cores, and OS Support

A plethora of emerging, specialized hardware, ranging from field-programmable gate arrays (FPGAs) to digital signal processors and multi-core processors, can significantly enhance performance by offloading computations. Such hardware is extensively used for tasks like cryptography and image processing, frequently accessing the local or global memory during execution [4]. Time constraints determine the choice of programming model, with user-programmable hardware, multithreaded execution, and multi-layered applications being common for concurrent tasks. Support for virtual memory enables accelerators to access user process memory via peripheral device interfaces, incorporating scheduling and resource management within the OS and eliminating the need for program alteration. Operating systems also manage accelerator execution, memory layout, and timing, facilitating integration into OS process-oriented systems. Defined as separate entities, hardware accelerators and specialized cores require OS support for effective usage. Acceleration through hardware models and programming concepts, like traditional approaches, allows existing applications to benefit without modification.

Multiprocessor integrated circuits with various structures, such as processors with multiple distributed cores, have proliferated in recent years. A lot of work has gone into creating programming, scheduling, and resource allocation methods, as well as the system and architecture support needed from the OS domain to build distributed and parallel systems. Pthreads and message passing interface (MPI) are two examples of basic programming methods and languages that have been created along with scheduling methods.

Scheduling and Resource Allocation in Heterogeneous Environments

Modern computing systems are characterized by heterogeneous hardware environments consisting of multi-core CPUs, various types of caches, and accelerators such as graphics processing units (GPUs) or FPGAs. This trend toward diversity arises from the increasing demand for high-performance, low-energy, and real-time systems. Although heterogeneous hardware demands novel OS support, existing studies show that research on scheduling and resource allocation in heterogeneous environments is insufficient [6]. Scheduling strategies that consider fairness or real-time requirements often experience serious performance degradation when more than two resources are involved, as well as application-level latency guarantees. Instead of seeking highly efficient solutions, the trend is towards distributed approaches to better manage multi-resource allocation owing to the growing number of cores, multi-core systems, and many-core configurations.

Operating System Architecture Diagram

Layered OS Architecture (Conceptual)

This diagram shows how the OS acts as a bridge between the hardware and users (Figure 1).

CONCLUSION

OSs play a vital role in connecting hardware and software by providing efficient resource management, security, and a user-friendly interface. This paper discusses the fundamental functions of OSs, including process scheduling, memory management, I/O handling, and system protection. It also highlights modern developments, such as virtualization, containerization, and heterogeneous computing, which are shaping the future of OS design.

In addition, this study presents practical examples and performance comparisons to demonstrate the impact of different techniques on system efficiency. The integration of hardware advancements with intelligent OS features remains an important research area. Despite significant progress, challenges remain in managing diverse hardware environments, improving energy efficiency, and ensuring security in distributed systems.

Future work should focus on developing adaptive and intelligent OSs that use artificial intelligence and machine learning techniques. These systems can improve automation and optimize resource allocation and enhance the overall system performance. Therefore, OSs will continue to evolve as key components of modern computing, supporting new technologies and user requirements.

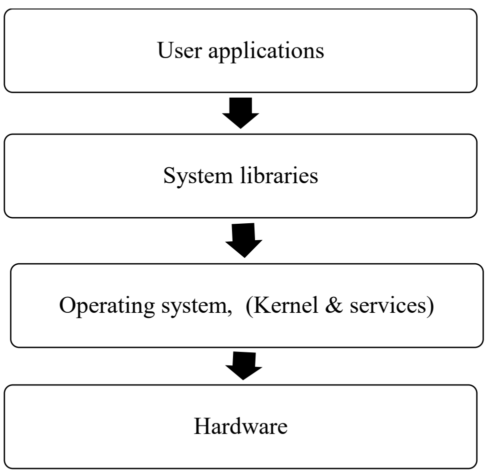

Figure 1. Operating system architecture.

REFERENCES

1. Vichare A. Intensional view of general single processor operating systems. [Preprint]. 2013. arXiv:1308.1199. doi:10.48550/arXiv.1308.1199.
2. Iliev AP. Formal description of components in operating systems. [Preprint]. 2014. arXiv:1402.4929. doi:10.48550/arXiv.1402.4929.
3. Farooq U, Iqbal MA, Nazir S. A glance into the future of human computer interactions. Int J Comput Sci Eng Appl. 2011;1:22–37. doi:10.5121/ijcsea.2011.1303.
4. Vuletic M. Unifying software and hardware of multithreaded reconfigurable applications within operating system processes [doctoral thesis]. Lausanne: École polytechnique fédérale de Lausanne (EPFL); 2006. 160 p. doi:10.5075/epfl-thesis-3626.
5. Yang T. Operating system support for modern applications [dissertation]. Amherst (MA): University of Massachusetts Amherst; 2009.
6. Seltzer MI, Endo Y, Small C, Smith KA. Issues in extensible operating systems. Cambridge (MA): Harvard University; 1997.
7. Nikseresht MR, Somayaji A, Maheshwari A. Customer appeasement scheduling. [Preprint]. 2010. arXiv:1012.3452. doi:10.48550/arXiv.1012.3452.
8. Weil F, Jamieson LH, Delp EJ. Dynamic intelligent scheduling and control of reconfigurable parallel architectures for computer vision/image processing. J Parallel Distrib Comput. 1991;13:273–285. doi:10.1016/0743-7315(91)90075-K.
9. Goel N, Garg RB. A comparative study of CPU scheduling algorithms. [Preprint]. 2013. arXiv:1307.4165. doi:10.48550/arXiv.1307.4165.
10. Cheng S, Higham L, Kawash J. Partition consistency: a case study in modeling systems with weak memory consistency and proving correctness of their implementations. Distrib Comput. 2014;27:363–389. doi:10.1007/s00446-013-0205-0.
11. Ge Z, Lim HB, Wong WF. Memory hierarchy hardware-software co-design in embedded systems. Cambridge (MA): Massachusetts Institute of Technology; 2005. Available from: http://hdl.handle.net/1721.1/7427.
12. Vallat B, Tauriello G, Bienert S, Haas J, Webb BM, Žídek A, et al. ModelCIF: an extension of PDBx/mmCIF data representation for computed structure models. J Mol Biol. 2023;435:168021. doi:10.1016/j.jmb.2023.168021.
13. Klimiankou Y. An enhanced multi-pager environment support for second generation microkernels. [Preprint]. 2014. arXiv:1404.1637. doi:10.48550/arXiv.1404.1637.
14. Gerber S, Zellweger G, Achermann R, Hoffmann M, Kourtis K, Roscoe T, et al. Cichlid: explicit physical memory management for large machines [preprint]. 2019. arXiv:1911.08367. doi:10.48550/arXiv.1911.08367.
15. Suh T, Blough DM, Lee HHS. Supporting cache coherence in heterogeneous multiprocessor systems. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition; 2004; Paris, France. Vol. 2. 2004. p. 1150–1155. doi:10.1109/DATE.2004.1269047.
16. Sensfelder N, Brunel J, Pagetti C. Modeling cache coherence to expose interference. In: Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS 2019); 2019; Stuttgart, Germany. Dagstuhl: Schloss Dagstuhl–Leibniz-Zentrum für Informatik; volume 133. 2019. pp. 18:1–18:22. doi:10.4230/LIPIcs.ECRTS.2019.18.
17. Wu HG. Fieldbus device drivers for accelerator control at DESY. [Preprint]. 2001. arXiv:hep-ph/0111257. doi:10.48550/arXiv.hep-ph/0111257.
18. Smith JM, Traw CBS. Operating systems support for end-to-end Gbps networking. Philadelphia (PA): University of Pennsylvania; 1993.
19. Soviani C, Edwards SA, Keromytis AD. Adding a flow-oriented paradigm to commodity operating systems. In: Proceedings of the First Annual Workshop on Interaction Between Operating System and Computer Architecture (IOSCA-1); 2005; New York, NY, USA. New York (NY): Columbia University; 2005. doi:10.7916/D80V8P56.
20. Yao Z, Talebi SMS, Chen M, Amiri Sani A, Anderson T. Minimizing trust with exclusively-used physically-isolated hardware. [Preprint]. 2022. arXiv:2203.08284. doi:10.48550/arXiv.2203.08284.

21. Payne BD. Improving host-based computer security using secure active monitoring and memory analysis [dissertation]. Atlanta (GA): Georgia Institute of Technology; 2010.
22. Yitbarek SF. Hardware mechanisms for efficient memory system security [dissertation]. Ann Arbor (MI): University of Michigan; 2018. Available from: https://hdl.handle.net/2027.42/147604.
23. Achermann R, Hossle N, Humbel L, Schwyn D, Cock D, Roscoe T. Secure memory management on modern hardware. [Preprint]. 2020. arXiv:2009.02737. doi:10.48550/arXiv.2009.02737.
24. Kiyanclar N. A survey of virtualization techniques focusing on secure on-demand cluster computing. [Preprint]. 2005. arXiv:cs/0511010. doi:10.48550/arXiv.cs/0511010.
25. Laadan O, Nieh J. Operating system virtualization: practice and experience. In: Proceedings of the 3rd Annual Haifa Experimental Systems Conference; 2010; Haifa, Israel. New York (NY): Association for Computing Machinery; 2010. p. 1–12. doi:10.1145/1815695.1815717.
26. Laadan O, Nieh J. Operating system virtualization: practice and experience. In: Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR '10); 2010 May 24–26; Haifa, Israel. New York (NY): Association for Computing Machinery; 2010. Article 17, 12 p. doi:10.1145/1815695.1815717.
27. Hanson HL. Coordinated power, energy, and temperature management [dissertation]. Austin (TX): The University of Texas at Austin; 2007.
28. Beloglazov A, Buyya R, Lee YC, Zomaya A. A taxonomy and survey of energy-efficient data centers and cloud computing systems. Adv Comput. 2011;82:47–111. doi:10.1016/B978-0-12-385512-1.00003-7.
29. Becker M, Chakraborty S. Measuring software performance on Linux. [Preprint]. 2018. arXiv:1811.01412. doi:10.48550/arXiv.1811.01412.
30. Foreman JC, Ragade RK, Graham JH. New software metrics for evaluation and comparison of advanced power management systems. IEEE Syst J. 2009;3:331–335. doi:10.1109/JSYST.2009. 2028747.
31. Andrade H, Crnkovic I. A review on software architectures for heterogeneous platforms. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan. 2018. p. 209–218. doi:10.1109/APSEC.2018.00035.

