JOURNAL OF EMERGING TECHNOLOGY AND DIGITAL TRANSFORMATION

ONLINE ISSN
3006-9726
PRINT ISSN
3006-9718

VOLUME . 4 ISSUE . 1 (2025)

PERFORMANCE, SCALABILITY, AND FLEXIBILITY: A COMPARATIVE ANALYSIS OF WINDOWS, LINUX, AND SOLARIS

Alishba Atta
Department of Computer Science, University of Southern Punjab Multan
alishbaatta17@gmail.com
Kinzul Eman
Department of Computer Science, University of Southern Punjab Multan
kinzuleman82@gmail.com
Boo ali Hassan
Department of Computer Science, University of Southern Punjab Multan
booalihassan5@gmail.com
***Muhammad Azam,**
Department of Computer Science, University of Southern Punjab Multan
Corresponding Author*:muhammadazam.lashari@gmail.com
Muhammad Zeeshan Haider Ali
Department of Computer Science, University of Southern Punjab Multan
ali.zeeshan04@gmail.com

RECEIVED	*ACCEPTED*	*PUBLISHED*
04 January 2025	17 February 2025	26 February 2025

ABSTRACT

All current computing systems are based on the functioning of operating systems (OS), which perform the functions of hardware abstraction, resource allocation, and process control. These various platforms of OS offer several options, and one is to compare the best and the most suitable to the particular architecture and use cases which demand a comparative analysis that delves into the mechanism and efficiency of operation. This paper gives a comparative review of three popular operating systems, Windows, Linux, and Solaris, in terms of important aspects like the process management system, file systems, threading system, memory management, virtual memory operations (page hits and misses), and kernel mode operations. Examine several peer-reviewed research articles and technical reports critically. This piece of work isolates the architectural distinctions, operational plans, and system-level innovations that mark these OSs. Windows focuses on ease of use and kernel modification; Linux focuses on modularity and open-source flexibility; while Solaris focuses on enterprise network scalability and high throughput. The results add a few helpful computer science contributions to the discussion of the design choices in operating systems and their results on the performance, stability, and resource utilization in various computing situations.

Keywords: *Operating Systems, Windows, Linux, Solaris, Process Management, File Systems, Threading Models, Memory Management, Virtual Memory, Kernel Architecture, Comparative Analysis.*

Introduction

Operating system is described as the underlying software layer which enables interaction among hard ware resources and software applications, and has taken a form which permits it to control important functions of a system including memory handling, process creation, file operations and device management. Its aim is to guarantee effectiveness in use of resources and stability, fairness, and security when carrying out various activities. Nonetheless, designing and implementing operating systems have long remained fraught with such challenges, among others, to have them efficiently multitask, protect their memory, coordinate I/O choices and optimize their performance on various hardware platforms (Papadimitriou & Moussiades, 2018).

Varying operating systems have made different approaches and architectural plans and mechanisms at the system level to tackle the challenges. An example is the transaction of Windows, which has a hybrid architecture of a kernel, a layered structure, subsystem isolation, fine-grained access control, and modular functionality, although this system can make complexities of low- level operations and create performance obstructions under some work loads (Vogels, 1999). However, Linux instead uses monolithic (but not modular) kernel architecture, and has been made flexible, able to be developed community-wide, and responsive to real-time tasks, particularly with regard to I/O-latency and interactivity tasks (Bovet & Cesati, 2000; Oliveira et al., 2022). Solaris, which has been designed to focus more on enterprise level systems, proposes project-based resource administration, threading at the kernel level and bundling in of the ZFS that all together facilitate the aspect of high reliability, scalability and efficient data management (Levy & Silberschatz, 1989; McDougall & Mauro, 2007).

The influence of these three systems demonstrates opposite points of view about OS development. whereas Windows focuses on user-friendliness, compatibility with old programs, Linus concentrates on transparency, user-configuration and speed. Solaris, in its turn, offers an enterprise type of framework that supports strong scheduling policies and enables advanced storage systems. As computing needs continue to change particularly in the fields of virtualization, cloud infrastructure and real-time analysis the background nature of the operating system becomes all the more crucial and the adequacy of design decisions will further determine the performance and stability of the system on a whole (Papadimitriou & Moussiades, 2018; Xu & Wang, 2024).

Kernel Mode

The kernel mode in **Windows** is a privileged level of execution on which core system services, device drivers, and the executive portion of the operating system execute, providing unrestricted access to system memory and hardware. Windows architecture operates on the hybrid kernel model: separating subsystems into a microkernel-like architecture, and combining it with the monolithic properties in terms of performance (Papadimitriou & Moussiades, 2018). The Kernel, Executive, Hardware Abstraction Layer (HAL), and Device Drivers are core components and they run in kernel mode, and inter-module communication is done by using dedicated interfaces. Although this structure is more stable and modular, the transition between user and kernel mode can create overhead where the system call or I/O is frequent (Vogels, 1999). Kernel-mode activities are secured using security mechanisms such as kernel patch protection and signed drivers, although any vulnerability in a privileged component can enable subversion of the whole system.

A monolithic kernel approach has been adopted in **Linux**, which implements most of the system services, such as device drivers, file systems, memory handling, in kernel mode and applications in user space communicate with these services through system calls. Kernel modules do (however) run in the privileged mode of the core kernel, so that both benefit and disadvantage remains as compared to the core kernel (Bovet & Cesati, 2000). High-performance interfaces of the system calls, like sysenter/sysexit or int 0x80, are often used to optimize the context switches betweenuser and kernel mode, preemption makes it possible to interrupt lower-priority tasks by higher- priority ones even within the kernel (Oliveira et al., 2022). Although Linux was able to offer flexibility due to modularity of its extensions and allow fast development of the kernel, high separation of kernel and user code is required to ensure reliability and security of the system.

Solaris has a monolithic modular kernel, with major services executing in the kernel mode but designed to enable them to be fine-grained controlled, configuration at run-time and fault containment. The kernel offers several scheduling classes, policies of managing resources, as well as a track of contract based services, and these policies are carried on in the kernel mode so that the system can be responsive and usage restrictions can be imposed (McDougall & Mauro, 2007). Solaris also enforces kernel preemption, meaning the lower-priority kernel threads can be preempted by the higher-priority ones thereby enhancing the real-time performance and elasticity to concurrent workloads (Papadimitriou & Moussiades, 2018). Kernel mode components will communicate using well organized interfaces, and the system calls are serviced using an efficient trap table mechanism which keeps an effective switch between user and kernel contexts. The reliability and modularity of operations on kernel mode is high in Solaris, but the complexity of interactions with these capabilities might make them more problematic to learn by developers and system administrators.

Process Management:

Process management has been adopted on a client-server model in Windows, whose implementation is based on a hybrid kernel form where subsystems of the user modes interrelate with the components of the kernel mode in an object-based abstraction. Every process is described by the executive object which includes process identifier, security structure, handle table and pointers to the thread structures. The creation of processes is associated with both Windows Subsystem and Windows Executive, and the child processes received many traits of parent processes, which are available in an organized duplication scheme (Papadimitriou & Moussiades, 2018). The task scheduling in Windows is priority-based and preemptive and is managed by a multilevel feedback queue; nevertheless, the system is highly configurable, having its ability to change the priorities between the threads according to the I/O wait times and execution behavior. That enables keeping responsiveness throughout the interactive tasks, but kernel switches and inter-process communication can have an overhead during the heavy workloads (Vogels, 1999; Xu & Wang, 2024).

On the contrary, Linux handles such processes using a monolithic kernel which is very simple but highly effective in performance as far as scheduling and resource allocation is concerned. All the processes in Linux are

monitored via task_struct structures which contain metadata
r
elated to the process such as process identifiers, CPU affinity, scheduling policy and memory mappings (Bovet & Cesati, 2000). Linux, the process management strategy addresses fairness and responsiveness with the Completely Fair Scheduler (CFS) where CPU time is apportioned fairly in accordance with the virtual run-time. Scheduling policies available in Linux are diverse, e.g. SCHED_FIFO,
SCHED_RR, SCHED_OTHER, and can be used to place real-time, time-sharing and the batch processes simultaneously on the system without conflicts in an efficient way. Copy-on-write techniques streamline forking mechanisms and the context switches of processes have been strongly reduced by the implementation of lightweight threads and shared memory models (Oliveira et al., 2022). The kernel supports multitasking well, however, in response to many active processes on a relatively small hardware, higher latency can be created.

Solaris has been designed to manage the process in a highly-scaled and resource-controlled fashion, as such it enhances large-scale and high-availability distribution. All of processes are incapsulated by the process structure, project ID, and the contract ID and Solaris introduce resource pools and projects to distribute and manage the system resources according to administrative policies (McDougall & Mauro, 2007). This system allows many scheduling classes as TS (Time-Sharing), FX (Fixed Priority), and RT (Real-Time) and supports flexible mapping between kernel threads and lightweight processes, which leads to efficient context switching and thread level parallelism. More qualified CPU limit, processor sets and resource usage per-process tracking facilities are also included in Solaris, allowing one to have a detailed control of the system behavior. They increase predictability and load balancing through these mechanisms and this is very useful in server and cloud workloads where resource isolation is important (Papadimitriou & Moussiades, 2018; McDougall & Mauro, 2007).

Threading Models

Windows has applied threads as fundamental units of execution to processes and then manipulated them by kernel-mode and user-mode scheduling mechanisms. All threads are created and handled through the Windows API and have a priority level, stack, and a unique thread ID that makes it possible to control execution, suspension and terminate (Papadimitriou & Moussiades, 2018). Windows kernel uses threads instead of processes to schedule and this permits much more control in terms of CPU utilization, particularly on applications that spawn multiple concurrent threads. Windows default threading model is a preemptive and priority driven model though fiber based threading can be applied to manually schedule user mode threads providing lightweight control which has fewer moments of context switching. But intra-thread communication usually needs synchronization primitives such as mutexes, semaphores and events which may, in turn, mean contention and complexity in situations of high concurrency (Vogels, 1999).

Linux supports the use of threading and the model adopted there is the POSIX threads (pthreads) model and threads are regarded as lightweight processes which are created using a system call, clone(). The threads have the same memory space, file descriptors and signal handlers of the parent process, which allows sharing information and may overcome the overhead cost of context switching (Bovet & Cesati, 2000). The Completely Fair Scheduler

(CFS) is a virtual runtime management method that focuses more on proportional fairness;scheduling the thread across the run time instead of enforcing a relative strict priority. Linux can run kernel-level threads as well as user-level threads, but the former are more popular because of their scalability and direct polling
by OS scheduler. Although threading in Linux is very efficient, some element of challenge can be observed and faced in parts of the thread-intensive applications in its capabilities in synchronization, CPU binding, and cache coherence in the multicore set-ups (Oliveira et al., 2022).

Solaris uses an exclusive two-order threading system, in which user-level threads are mapped to kernel-level Lightweight Processes (LWPs), and this support offers additional freedom to control parallel execution (McDougall & Mauro, 2007). Both the user-level library and the kernel schedule threads and thus multiple user threads can be active simultaneously on accessible processors. Solaris has also introduced the concept of bound threads, so that a user thread never gets disassociated with a LWP again once bound, and an application with a deployment thus has predictable latency-sensitive execution. This model reduces overhead in the creation and destruction of threads, and this model improves scalability of multithreaded applications. Also, Solaris provides multiple scheduling classes andreal-time threads and can schedule the critical work-load with no starvation of the less important processes (Papadimitriou & Moussiades, 2018). Although two-level model is more effective in control and performance, it may bring complexity to the thread management as new layer of abstraction is introduced

Memory Management

Memory management in Windows has taken the form of layered implementation that differentiates between paged and non-paged memory pools and renders an abstract virtual address space presentation to every process coupled with the provision of an effective kernel-mode processing of memory. Practical memory monitoring is through Page Frame Number (PFN) catalogs, and it is the obligation of the Virtual Memory Manager (VMM) to allocate, reserve, and decommit memory pages dependent on the arrangement of use and application requirements (Papadimitriou & Moussiades, 2018). Windows also has capabilities on memory mapped files, shared memory, and heap management and protection of the memory takes place through access rights and page level-permissions. Garbage collection on managed applications is also available as part of the system and the Memory Manager is used to check working sets and to prune them when the memory pressure is observed. Although the structure of memory management in Windows is very organized, memory-intensive applications challenge memory through fragmentation and overheads to internal data structures (Vogels, 1999; Xu & Wang, 2024).

Linux employs zone-based memory anatomy, that is, it categorizes the physical memory into various zones, including ZONE_DMA, ZONE_NORMAL, and ZONE_HIGHMEM, and allocates each particular zone separately in order to serve devices that have varying addressing capability (Bovet & Cesati, 2000). The buddy system and the slab allocator are used to allocate large and frequent small-sized allocations to kernel functions respectively, and these methods are to minimize the fragmentation and accelerate reusing memory. Moreover, the Linux Kernel provides page cache, slab cache and swap mechanisms, which are collaborated to police the meter between performance and memory consumption. Protection and isolation of processes in their memory is provided by page tables implemented in hardware, through an MMU. Despite Linux's excellent efficiency and modularity, some workloads can become caught in latency spikes due to memory strain or invalidation solicitation (Oliveira et al., 2022).

Solaris has a more modular and dynamic attitude toward memory management, and allows high- end functions, including memory capping, resource pools, and the capability to dynamically reconfigure against application changes. Virtual Address Cache (VAC) and Unified Page Cache manage physical memory by causing a reduction in redundancy between file system cache and anonymous memory (McDougall & Mauro, 2007). It has a slab allocator on kernel memory and memory resources can be divided per project or zone, which enhances both predictability and performance isolation on multi-tenant systems. The Solaris operating system also provides tools in memory accounting and real-time usage monitoring, which helps administrators to modify the memory behaviour and find bottlenecks. However, though Solaris is tuned towards high-throughput and mission-critical environment deployment, it can become harder under the heterogenous or unpredictable workloads situation (Papadimitriou & Moussiades, 2018).

Virtual Memory

Virtual memory implementation in windows has taken the form of a demand paged system with each process being assigned a virtual address space shared privately with all other allocation and loading of memory pages to physical memory only on demand. In the case of a page reference, where a page is read in memory, it leads to a page hit when the next access has a minimum delay but in the case of a page fault (page miss), the memory manager realizes there is a page fault and reads the requested material cold in the disk into RAM (Papadimitriou & Moussiades, 2018). To maintain the set of pages actively used by each process, working sets in windows measure the active pages of each process and an up-threshing mechanism is activated when the pressure on the memory grows to cut off the least popular pages. Some other types of list present in the system include the standby list and the modified list that are useful in reclaiming the memory effectively without discarding reusable pages as soon as they arise. Whereas Windows virtual memory subsystem is engineered in a manner that optimizes throughput, it is possible to add latency on a system with less RAM or very high level of I/O (Vogels, 1999).

In Linux, the implementation of virtual memory consists of an integrated scheme of paging, demand-loading and memory extension by means of swap, and process i.e. the individually allocated virtual address spaces is managed by Memory Management Unit (MMU) and kernel data structures including mm_struct (Bovet & Cesati, 2000). Page hits are managed immediately by the Translation Lookaside Buffer (TLB), and page faults invoke the page fault handler to make a decision on whether or not to allocate a new page, read in an existing page or to kill the process with an invalid access. The Least Recently Used (LRU) algorithm with aging that Linux uses in paging replacement and a swap area, respectively, when physical memory is exhausted. Page faults, hits, and memory pressure can be monitored in the real-time using vmstat utility and /proc/meminfo interface. Although, the system runs optimally on normal workloads, too many paging or thrashing can break down performance on memory demands that are more than the physical space available (Oliveira et al., 2022).

The Solaris virtual memory subsystem is closely coupled with file system and I/O management layers and engineering focuses on ensuring scalability, fault tolerance and efficient page management in an enterprise-class system. Solaris allocates memory anonymously both for heaps and stacks and file backed regions are mapped and distinguished in page fault processing. The problem of page hits is solved quickly with the help of the layered cache hierarchy, such as Adaptive Replacement Cache (ARC) in the system where ZFS is used, which enhances memory re-use and avoids the useless disk I/O (McDougall & Mauro, 2007). Page misses are controlled by predictive paging and the system also has large pages and shared memory segments, which improves performance of applications like databases. Solaris also monitors memory in a per-process basis and has dynamic page scanner threads which keep a balance in the memory by discarding pages which are not used frequently. The design eliminates paging overhead, offers continuous response time especially in a load that is high in memory throughput and low latency (Papadimitriou & Moussiades, 2018).

File Systems

The file system architecture of Windows has focused on NTFS (New Technology File System), first introduced to overcome the restrictions of FAT, to make available tomorrow such features as file system journaling, access control lists (ACLs), and file system metadata tracking. NTFS is designed and organized as a metadata intensive system, this is, almost all data such as information on files and hierarchy of directories is stored as files itself in the Master File Table (MFT) (Vogels, 1999). This structure enables fast access to the attributes of files and has the capability of supporting advanced features including encryption, compression and disk quota. NTFS has also included a change journal, which keeps track of the changes to enhance its recovery activities and synchronization. Although robust, metadata procedures and journaling costs can create performance bottlenecks when big file operations or composite I/O are undertaken (Xu & Wang, 2024).

Linux has numerous file systems however, ext4 is the most commonly supported as it is stable, backward compatible and is faster than other ext file systems. More so it reduces fragmentation through allocation of contiguous blocks intended to store files (login, 2007). Also on the list is delayed allocation, persistent preallocation and checksumming of the journal that not only enhances reliability but also enhances write performance. More than ext4, Linux possesses modern file systems like Btrfs and XFS that have volume management, snapshotting built-in, and on- demand inode allocation of enterprise-level workloads. Such flexibility enables the administrators to select a file system on a workload basis, but compatibility and tuning requirements could raise the complexity of the configuration (IRJET, 2021).

ZFS replaces the logical volume manager, and it applies such copy-on-write (COW) techniques that will guarantee that, even when computer crashes are experienced, given issues of data integrity are at no risk (McDougall & Mauro, 2007). One of the most significant features of ZFS is Adaptive Replacement Cache (ARC) that offers the efficient block-level caching and boosts the read/write throughput in large systems. ZFS also provides end-to-end checksumming, redundancy via RAID- Z, and data healing in automated mode to prevent undetectable data corruption. Even though ZFS uses more memory than the simpler file systems, it provides unrivaled scalability, which is why it is perfect when used in high-availability servers and on storage-intensive applications (Levy & Silberschatz, 1989; Papadimitriou & Moussiades, 2018).

Table:

Component	**Windows**	**Linux**	**Solaris**	**Our Suggestions**
Kernel Mode	Hybrid kernel	Monolithic kernel	Modular monolithic kernel with kernel preemption	Solaris has modular control and preemption; Linux is flexible; Windows is a compromise between usability and control.
Process Management	Object-oriented process structures	task_struct and Completely Fair Scheduler (CFS)	Resource pools and scheduling classes	Solaris is suited for enterprise workload control; Linux is responsive; Windows has detailed object-based management.
Threading Models	Kernel-mode threads and fibers	POSIX threads using clone()	Two-level model with Lightweight Processes (LWPs)	Solaris supports scalable threading; Linux is optimized for performance; Windows offers flexible threading with fibers.

Memory Management	Paged/non-paged pools and PFN database	Zone-based management with slab and buddy systems	Slab allocator and memory resource pools	Solaris allows fine-grained control; Linux is performance-efficient; Windows is organized but prone to fragmentation.
Virtual Memory (Page Hit & Miss)	Working sets and page trimming	Swap and Least Recently Used (LRU)	Predictive paging with ARC in ZFS	Solaris excels with ARC cache; Linux has a balanced strategy; Windows may underperform with limited RAM.
File Systems	NTFS with journaling and MFT	ext4, Btrfs, XFS	ZFS with Copy-on-Write, ARC, and checksumming	ZFS in Solaris is ideal for high-integrity storage; Linux supports multiple file systems; NTFS in Windows is secure and backward-compatible.

Discussion

Comparative analysis of Windows, Linux and Solaris operating systems highlights some of the major architectural and functional differences that really affect the performance, flexibility, and scalability of the system. Every OS uses varying approaches in the execution of such core elements as the kernel, memory, the processes and threads, file systems and virtual memory. These distinctions conform to each system design philosophies and intended environments by users. Under kernel architecture, Windows uses a hybrid design, balancing usability and performance, whereas Linux resorted to a monolithic kernel model which is efficient and reflects transparency. Solaris is characterized by a monolithic-kernel with modularity and kernel preemption which improves control and responsiveness of high-performance in enterprise applications. There are also wide variations in the process management techniques. Windows applies object-oriented process structures which are compatible with a more object-oriented structure and architecture. The Linux task_struct structure and Completely Fair Scheduler makes it responsive and fair and thus attractive to general-purpose computing. Solaris however offer advanced process control in terms of resources pools and scheduling classes, making it the best in multi user and enterprise worlds where work segments and control is necessary. In the case of threading models, the two-level scheme of Solaris based on the use of LWPs is remarkable due to its approach to scalability, so the two-level scheme is ideal to multitasking systems heavy. Linux uses performance-optimized POSIX threads, but Windows supports both kernel threads and lightweight fibers. This has made Windows flexible to different possible applications but not particularly effective on thread scalability unlike Solaris. The memory management systems also further distinguish these operating systems.

Linux follows a zone scheme along with slab and buddy allocators, which are low overhead and very efficient.Windows uses paged and non-paged pools with a PFN database and delivers well structured memory management though at risk of fragmentation. Solaris once again targets enterprise requirements, with slab based allocation with resource pools to provide detailed control and optimised performance. Even though in the sphere of virtual memory Solaris came first with its ARC integrated predictive paging via ZFS, which has better ability to manage cache. Linux provides a fair technique in manageable space with the use of swap and LRU strategies, whereas Windows provides more intensive consideration in functioning sets and cutting strategies, which can deteriorate with a limited RAM condition. Such disparities indicate Solarisas an effective option in applications that are in demand of heavy memory applications. Lastly, Solaris has support within the file system with its ZFS that is known to have a high integrity, checksumming, and Copy-on-Write architecture. Linux supports a large variety, as it has ext4, Btrfs and XFS that enable it to adapt to numerous applications. Windows has not yet moved on to NTFS, and remains backward-compatible and secure, although not nearly as scalable or fault-tolerant as ZFS. In general, the comparison reflects that Solaris is ideal in scenarios that entail enterprise level, high throughput, modular control and stability. Linux takes advantage of high flexibility and performance in development and general purpose tasks compared to Windows that is user-friendly with well-organized management, therefore suitable to the environment of personal computers and administration.

Conclusion

The above review has critically analyzed the architectural and operational difference between Windows, Linux, and Solaris operating systems, key examples of architecture and operation features include kernel modal, process and memory control, threading style, file system and virtual memory protocol. All systems have their own strengths closely related to their design philosophy and the intended field of application. Windows is characterized by an orderly and customer- friendly system where object-oriented management and compatibility is possible with corresponding NTFS, a system that is appropriate to the administration and desktop. Linux, which has a monolithic kernel, flexible threading, and effective memory plans, is a good option among developers and general-purpose computing. Solaris, being developed with modularity, kernel preemption, and state-of-the-art creates such as ZFS and ARC in mind and is especially geared towards large-scale high-integrity systems which require both scale and the ability to gain and exercise a fine-grained level of control. In practice, none of the operating systems tops every single metric; it is all about context when it comes to the best fitting option. At the enterprise-level, Solaris is light-years ahead, Linux is the best regarding performance and flexibility, and windows offer a happy medium and an easy way to do organized work. Such comparative perception allows making informed choices in system designing, deployment, and optimization within the various computing landscape.

References

1) Anderson, T. E., Bershad, B. N., Lazowska, E. D., & Levy, H. M. (1991). Scheduler
2) activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS), 10*(1), 53–79.
3) https://doi.org/10.1145/103727.103729
4) Badam, A., Kashyap, S., Chidambaram, V., Prabhakaran, V., & Gunawi, H. S. (2019).
5) Optimizing systems for persistent memory with SplitFS. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 19)*, 105–120.
6) Druschel, P., & Kaashoek, M. F. (1996). A comparison of mechanisms for efficient
7) communication in multiprocessor operating systems. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI '94).*
8) Kahanwal, B. S. (2013). Classification and comparison of file systems. *International Journal of Advanced Research in Computer Science and Software Engineering, 3*(1), 295–300.
9) Klein, M., Rajkumar, R., & Lehoczky, J. P. (2000). A partitioning operating system for predictable execution and isolation. In *Proceedings of the IEEE Real-Time Systems*
10) *Symposium (RTSS)*, 231–242. https://doi.org/10.1109/REAL.2000.896049
11) McKenney, P. E., & Slingwine, J. D. (1998). Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS)*, 509–518.
12) POSIX.1c-1995. (1995). *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API), Amendment 1: Realtime Extensions [C Language]*. IEEE Std 1003.1c-1995.

13) Rajkumar, R., Sha, L., Lehoczky, J. P., & Zelenka, J. (1998). Resource kernels: A

14) resource-centric approach to real-time and multimedia systems. In *Proceedings of SPIE Multimedia Computing and Networking Conference*, 150–164.

15) Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

16) Klein, J., & Seltzer, M. (1999). The interrupt and process models: Performance and flexibility for high-speed networking. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

17) Zadok, E., & Nieh, J. (2000). FiST: A language for stackable file systems. In *Proceedings of the USENIX Annual Technical Conference*, 55–70.

18) Sipek, J., Pericleous, Y., & Zadok, E. (2007). Kernel support for stackable file systems.

19) *Proceedings of the Ottawa Linux Symposium*, *2*, 223–227.

20) Zadok, E., Badulescu, I., & Shender, A. (1999). Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, 57–70.

21) Narayan, S., Mehta, R. K., & Chandy, J. A. (2010). User space storage system stack modules with file level control. In *Proceedings of the 12th Annual Linux Symposium*, 189–196.

22) Rosenthal, D. S. H. (1990). Evolving the vnode interface. In *Proceedings of the USENIX Technical Conference*, 107–118.Rosenthal, D. S. H. (1992). Requirements for a “stacking” vnode/VFS interface (Technical Report SD-01-02-N014). UNIX International.

23) Zadok, E., Iyer, R., Joukov, N., Sivathanu, G., & Wright, C. P. (2006). On incremental file system development. *ACM Transactions on Storage (TOS)*, *2*(2), 161–196.

24) https://doi.org/10.1145/1149964.1149967

25) Kahanwal, B. S., Singh, T. P., & Tuteja, R. K. (2011). Performance evaluation of Java File Security System (JFSS). *Advances in Applied Science Research*, *2*(6), 254–260.

26) Kahanwal, B. S., Singh, T. P., & Tuteja, R. K. (2011). A Windows-based Java File

27) Security System (JFSS). *International Journal of Computer Science & Technology*, *2*(3), 25–29.

28) Kahanwal, B. S., Singh, T. P., Bhargava, R., & Singh, G. P. (2012). File system – A component of operating system. *Asian Journal of Computer Science and Information Technology*, *2*(5), 124–128.

29) Idris, A., Aliyu, A. A., & Muhammad, U. S. (2022). Comparative analysis of modern operating systems. *Nigerian Journal of Computing, Engineering and Technology (NIJOCET)*, *1*(2), 50–64.

30) Papadimitriou, S., & Moussiades, L. (2016). A comparative evaluation of core kernel features of the recent Linux, FreeBSD, Solaris and Windows operating systems. In *Proceedings of the World Congress on Engineering (WCE), London, UK.*

31) Vogels, W. (1999). File system usage in Windows NT 4.0. *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, 93–109.

32) https://doi.org/10.1145/319344.319158

33) Nikhil, K., Hariprasad, S. A., & Aditya, B. (2021). Comparative study on various file system implementations on different OS. *International Research Journal of Engineering and Technology (IRJET)*, *8*(11), 1186–1191.

34) Ding, Y., Bolker, E., & Kumar, A. (2003). Performance implications of hyper-threading.

35) *Proceedings of the Computer Measurement Group's International Conference.*

36) Nakajima, J., & Pallipadi, V. (2002). Enhancements for hyper-threading technology in the operating system. *2nd Workshop on Industrial Experiences with Systems Software.*

37) Lyerly, R., Barbalace, A., Jelesnianski, C., Legout, V., Carno, A., & Ravindran, B. (2016). Operating system process and thread migration in heterogeneous platforms. *MARS 2016 Workshop.*

38) Draves, R. P., Bershad, B. N., Rashid, R. F., & Dean, R. W. (1991). Using continuations to implement thread management and communication in operating systems. *ACM*

39) *SIGOPS Operating Systems Review, 25*(5), 122–136. https://doi.org/10.1145/121132.121155

40) Kumar, N., & Shukla, S. (2015). A comparative evaluation of core kernel design

41) approaches. *International Journal of Scientific and Research Publications, 5*(7), 1–6.

42) Kumar, A., & Singh, R. (2020). Memory management techniques in operating systems: A survey. *International Journal of Future Generation Communication and Networking, 13*(3), 1138–1145.

43) Ehsan, M., & Hussain, S. (2019). Performance analysis of different file systems in Linux environment. *International Journal of Innovative Technology and Exploring Engineering (IJITEE), 8*(11), 1190–1194.

44) Dandamudi, S. P. (2005). *Fundamentals of computer organization and design.* Springer.

45) Mogul, J. C. (1989). The association of system performance with UNIX memory management. *USENIX Conference Proceedings*, 1–15.

46) Sharma, R., & Tiwari, A. (2016). Survey of various process scheduling algorithms.

47) *African Journal of Computing & ICT, 9*(1), 1–7.

48) Jain, P., & Nema, R. K. (2015). Real-time micro-kernel threads-based operating system (MiThOS). *International Journal of Engineering Research and Applications (IJERA), 5*(4), 81–86.

49) Mahajan, A., & Sood, M. (2014). Real-time OS for wireless sensor networks: An event- driven multithreaded approach. *International Journal of Embedded Systems and*

50) *Applications, 4*(3), 1–8.

51) Kumar, V. (2016). File system design and implementation techniques: A survey.

52) *International Research Journal of Engineering and Technology (IRJET), 3*(7), 1–6.

53) Goyal, M., & Singh, S. (2016). Role of process and memory management in Windows, Linux, and Mac OS. *International Journal of Advanced Research in Computer Science, 7*(6), 100–106.

54) McKusick, M. K., & Neville-Neil, G. V. (2004). *The design and implementation of the FreeBSD operating system.* Addison-Wesley Professional.

55) Tanenbaum, A. S., & Bos, H. (2015). *Modern operating systems* (4th ed.). Pearson.