A Survey on Microkernel Based Operating Systems and Their Essential Key Components

Preprint *in* SSRN Electronic Journal · June 2023

DOI: 10.2139/ssrn.4467406

2 authors:

Md. Ratan Rana
American International University-Bangladesh
6 PUBLICATIONS **3** CITATIONS
SEE PROFILE

Saikat Baul
American International University-Bangladesh
6 PUBLICATIONS **3** CITATIONS
SEE PROFILE

A Survey on Microkernel Based Operating Systems and Their Essential Key Components

1st Md. Ratan Rana
Dept. of Computer Science
American International University-Bangladesh
Dhaka, Bangladesh
ratanrana9684@gmail.com

2nd Saikat Baul
Dept. of Computer Science
American International University-Bangladesh
Dhaka, Bangladesh
saikatbaul.cs@gmail.com

***Abstract*—In a conventional monolithic operating system architecture, the kernel delivers all the necessary services to the application programs. As the requirements of the operating system increase, the kernel expands in size and increases complexity. The introduction of Operating Systems (OSs) focusing on Microkernels was due to the difficulties mentioned above caused by Operating Systems with Monolithic kernels. Operating systems based on microkernels offer enhanced security and flexibility to the system. This research presents an in-depth review of eleven unique operating systems based on a microkernel architecture. The study focuses on three key factors: inter-process communication (IPC), memory management, and process scheduling. This review offers insights into the various trends observed in the design of Microkernel Based Operating Systems. The data sources included books, research papers, and official documentation of individual microkernels.**

***Index Terms*—Operating system, Microkernel, Process scheduling, Memory management, Inter-process communication**

I. Introduction

Microkernels emerged as a response to the issues that arose from monolithic kernels [1]. The microkernel-based operating system has gained popularity recently. The fundamental basis of a microkernel is a highly efficient and compact kernel that provides only the most essential operating system functions. Operating systems based on microkernel technology provide several benefits over conventional monolithic kernels, including enhanced security, modularity, and adaptability [2]. These operating systems can be expanded with additional services without needing modifications to the grain, and they can be more readily adjusted to various hardware architectures. The kernel's reduced size and more straightforward design simplify checking for security and accuracy.

The separation of kernel and server affords to safeguard within the operating system itself, a feature that assumes greater significance as operating systems offer a more comprehensive range of services. Utilizing the microkernel approach facilitates the coexistence of diverse operating systems which can operate atop the microkernel. The Windows NT operating system developed by Microsoft can support operating system interfaces of both OS/2 and POSIX [3]. Various sources evidence this. Hence, diverse applications with varying requirements may opt for the operating system that is most appropriate for their specific needs. The attribute of modularity facilitates the convenient dissemination of the operating system by allocating servers on diverse computing nodes, subject to the availability of an adequately proficient IPC mechanism.

The objective of this article is to examine the fundamental principles of microkernels. This research investigates various iterations of microkernels across different generations. We examine various microkernel-based operating systems arranged systematically, RC 4000 Nucleus, StarOS, CHORUS, QNX, SPIN, Exokernel, L4, L4re, seL4, NOVA Micro hypervisor, and Muen Separation Kernel about their characteristics and think about the concept of a microkernel alone with their process scheduling, memory management, and IPC.

Afterward, the rest of the document will be presented in the following manner: Section 2 provides a comprehensive summary of relevant research that other scholars in the field have contributed. Section 3 explores various microkernels, whereas Section 4 focuses on how individual microkernels have addressed three key features essential to all microkernels. Finally, Section 5 provides a concluding statement and recommends possible directions for future research.

II. Related Work

The effectiveness of three different microkernels was examined [4]. Amoeba, Mach, and Chorus are among the microkernels that have been evaluated. The assessment encompassed analyses of microkernel memory management, process management, and communication, focusing on identifying similarities and differences. The study noted that despite being developed by distinct groups, the three microkernels share numerous similarities in their implementation. The analysis focused on the evolution of the L4 microkernel over two decades [5]. A comparative analysis was conducted between the design specification implemented in the initial iteration of L4 by its creator Liedtke, and the contemporary L4 microkernel. The design principles of the L4 microkernel were deliberated, which include minimality, inter-process communication (IPC), user-level design drivers, and resource management. The article additionally explained the design choices and execution of the IPC, resulting in enhanced speed. The abovementioned factors encompass the implementation language, non-standard calling convention, lack of portability, direct process switch, preemption, lazy scheduling, rigorous process orientation, and virtual thread control block (TCB) array. The seL4 operating system kernel was subject to a thorough investigation, which has advanced the L4 model to its greatest extent. It was the first kernel that was formally validated and thoroughly tested for worst-case execution times. The study has shown that despite changes, the core principles of minimalism, universality, and efficient IPC are the primary factors influencing design and implementation choices.

In [6], the study overviews seven operating systems based on microkernel architecture. The current analysis offers an understanding of the diverse patterns observed in the design of microkernel-based operating systems. The findings indicate that there has been no significant deviation in the fundamental tenet of minimality and how microkernel operating systems execute their IPC, memory management, and scheduling mechanisms. The research evaluated eight operating systems developed using a microkernel architecture [3]. This study involved arranging microkernel operating systems chronologically to facilitate a comparative analysis of their features and examine the microkernel concept's evolution in each scenario. Matarneh introduced a model for multi-core operating systems that utilizes a multi-microkernel approach for system processing instead of a single-microkernel course [7]. The multi-microkernel operating system is partitioned into two distinct components: master kernel and slave kernels. The primary microkernel will take responsibility for the essential functions within the system. Concurrently, the slaves will take responsibility for associated tasks within the framework, such as facilitating correspondence between the two separate procedures.

Memory management was highlighted as one of the critical roles of an operating system by the author [8]. This function is agnostic to the type of kernel architecture. Typically, Operating Systems initiate requests for physical memory allocation (PMA) during the boot-up process. The PMA requests are generated by software programs that dynamically operate within the operating system. The kernel-mode PMA typically manages and reserves less than 8% of the physical memory. The user-mode PMA handles the remaining portion. As per the findings of [9], the memory is partitioned into self-governing segments allocated to processes and threads during the system's run-time. In [10], the author defined two primary categories of inter-process communications: Shared memory: Creating a chunk of memory, and distributing it across associated programs. These processes then communicate separately by reading from and writing to the shared memory area. This works well for exchanging large amounts of data and is quicker than message forwarding since it uses fewer system calls. Message passing: For this to happen, the processes must communicate directly. Since the methods communicate directly with one another and do not share the same address space, this works well with small amounts of data and eliminates the need to prevent conflicts. In contexts involving distributed computing, these advantages are the most notable.

[2] examines microkernel technology from various angles, including its architecture, background, characteristics, and potential applications. This article outlines the future research directions of the microkernel from interoperation, distributed storage, security, and other areas considering the shortcomings of the microkernel design in the future connectivity of everything. Experiments based on the seL4 microkernel are addressed. These experiments include virtualization and performance evaluations against Linux.

III. Overview of microkernel

This review paper selected microkernels for evaluation from [11] and [12] to provide a balanced representation of microkernels ranging from research-only to production-level microkernels. Based on the availability of research articles, eleven different microkernels were chosen from the two lists above for evaluation. The microkernels under consideration are RC 4000 Nucleus, StarOS, CHORUS, QNX, SPIN, Exokernel, L4, L4Re, seL4, NOVA Micro-hypervisor, and Muen Separation Kernel. This section aims to present a comprehensive summary of the microkernels that have been listed.

A. RC 4000 Nucleus

Hansen initially presented the nucleus for the RC 4000 system in 1970 [13]. Hansen's research centered on the fundamental principles of managing a system of parallel and interdependent processes. The result led to the development of the Nucleus system, which was explicitly designed to assist in process abstraction and ICP. It has been acknowledged that the systematic expansion of the system necessitates the incorporation of additional attributes beyond the nucleus through the operation of system processes.

B. StarOS

StarOS is an operating system designed to operate on the Cm* multiprocessor at Carnegie-Mellon University. Its development can be traced back to the year 1977 [14]. The Cm* multiprocessor utilized in StarOS is comprised of 50 processors that are arranged in clusters of 10. The clusters exhibit interconnectivity through a local bus and are linked through an inter-cluster bus. Although individual processors possess restricted memory, the overall system has adequate memory to support a complete operating system and its associated applications. Individual processors' limited memory capacity necessitates the operating system's distribution across multiple processors. Each processing unit performs the execution of a central processing unit and a distinct subset of the modules within the operating system. Each module's composition consists of multiple processes, albeit with StarOS processes that are relatively less complex and minor than traditional methods.

C. CHORUS

The beginning of CHORUS was instigated at INRIA, a French research institute, in 1980. The current system being analyzed represents the third version, developed in 1987 [15]. The CHORUS system is a distributed computing infrastructure consisting of a moderate-sized microkernel, referred to as the nucleus, and a set of system servers that provide standard operating system functionalities. Among these servers is a group that furnishes Unix system services. The nucleus is responsible for critical functions such as interrupt handling, scheduling, virtual memory management, and inter-process communication. The thread serves as the fundamental unit of execution, encapsulated within an actor that functions as the unit of resource ownership, like a process in modern terminology.

D. QNX

QNX is a proprietary real-time operating system based on a microkernel architecture that its developer has marketed as a well-executed implementation of the microkernel design. The inception of the subject matter occurred in 1981, while the iteration expounded upon in this context is the contemporary rendition, namely version 4.0 [16]. The microkernel architecture enables IPC, process scheduling, interrupt dispatching, and network communication. The ultimate offering may seem inconsistent: most minimal microkernels integrate network services as autonomous servers. Indeed, the provision of such services is facilitated by the availability of a Network Manager. However, it can also connect to the microkernel, thereby enabling the realization of network-transparent IPC. The feature mentioned above allows seamless network transparency across all system levels, facilitating an application's utilization of a server on a different network node.

E. SPIN

In 1994, the University of Washington proposed the SPIN operating system [17]. The text acknowledges the potential of microkernels in delivering operating system services customized to specific applications. The present methodology of an application's interaction with a user-level server is considered suboptimal owing to the considerable overhead incurred in traversing from the application to the kernel, then to the server, and subsequently back to the kernel, culminating in the return to the application. As stated in [18], relocating the server to the kernel space in the Mach microkernel resolved the issue by eliminating 50% of the overhead. However, this approach weakens the protection of the operating system, which is one of the highly desirable characteristics of microkernels, in exchange for a level of performance that is on par with that of monolithic systems.

F. Exokernel

The idea of a minimal microkernel, commonly called an "exokernel," was first introduced by the Massachusetts Institute of Technology (MIT) and released in 1995 [19]. The fundamental aim of an exokernel is to provide a simplistic hardware abstraction layer that enables the safe and efficient allocation of resources among applications at the user level. The applications mentioned above are operating systems developed on the Exokernel platform with a comprehensive range of functionalities. It exports hardware resources instead of emulating them and does not aim to provide any abstraction. The efforts to achieve a minimalist goal result in developing a thin layer placed above the hardware. This layer provides applications with a secure mechanism to connect to resources, a protocol to withdraw resources, and an interface to transfer control.

G. L4

German National Research Center for Information Technology developed L4 in 1995 [20]—this microkernel, belonging to the second generation, aimed to revitalize the microkernel movement. The matter of performance was tackled by optimizing the interface provided by the microkernel. The aim of L4, on the other hand, was comparatively moderate. The microkernel architecture facilitated the implementation of threads and address spaces while enabling IPC between these distinct address spaces.

H. L4Re

The L4re microkernel represents one of the later versions of the original L4 microkernel, which has been subject to ongoing development efforts since 1997. The system follows the design principles of the L4 microkernel family. Any component residing within the kernel space is necessary for the system's proper functioning and cannot be relocated to the user space without compromising system functionality [21]. The Runtime Environment comprises the Fiasco—OC microkernel, its lowest-level and highest-privileged program. The system in question has the characteristic of excluding intricate services from its internal structure and solely consists of kernel objects—the integration of Fiasco.OC microkernel and L4re services yield a comprehensive L4re microkernel that encompasses all essential services for the execution of user-level applications. The L4re microkernel has demonstrated scalability across various systems, from embedded to high-performance computing (HPC) systems. Consequently, it has been utilized in multiple domains, including security, mobile devices, and automobile technology.

I. seL4

The NICTA group developed a third-generation microkernel called seL4 in 2006. The primary objective was to establish a foundation for highly secure and dependable systems that meet security standards, such as Common Criteria and others, through a design created from the ground up [22]. The microkernel in question is distinguished by its exclusive guarantee, which comprises machine-checked proofs that validate the functional correctness of the executable binary implementation against a formal model. Additionally, the standard model is designed to enforce integrity and confidentiality, except for timing channels. Like other security-focused systems, seL4 employs capabilities to regulate access control.

J. NOVA Micro-hypervisor

The start of the NOVA Micro-hypervisor can be traced back to the research report published by the Technical University of Dresden in 2006. The micro-hypervisor in question is denoted by a recursive acronym, which expands to NOVA OS Virtualization Architecture [23]. NOVA is a hybrid system that integrates the functionalities of a microkernel and a hypervisor. The system offers a compact trusted computing base (TCB) that caters to end-user applications and virtual machines. The system's TCB also offers virtualization, communication, scheduling, and resource management functionalities. The virtualization architecture provided by the NOVA micro-hypervisor is deemed secure owing to its lightweight components that can be designed and developed independently and verified for accuracy [23].

K. Muen Separation Kernel

The Muen Separation Kernel is a microkernel with specialized features that have undergone formal verification, thereby displaying no evidence of defects in its source code during execution. This represents a novel and unprecedented occurrence. In addition, as a separation kernel, it provides a platform where multiple components can operate and communicate with each other according to a predetermined policy. Failure to comply with the policy will result in the execution of said components in isolation. The potential of side and covert-based channel attacks is constrained. In addition, it can be observed that the Muen Separation Kernel exhibits a comparatively reduced size and a greater degree of staticity when compared to dynamic microkernels. This facilitates the implementation of formal verification. This enables the application of formal verification techniques [24]. The Muen Separation Kernel has been employed as the foundation for complex security solutions currently under development [24].

IV. Components of Microkernel

This review paper focuses on how individual microkernels have addressed three key features essential to all microkernels: process scheduling, memory management, and inter-process communication. This section provides precise details regarding managing the essential features of the eleven listed microkernels.

A. Process Scheduling

The execution of processes within a system can be traced back to the operating system at its inception, wherein a parent-child relationship is established. Parents can initiate and terminate child processes and generate and eliminate them. The previously mentioned facilitates the execution of program swapping and loading beyond the nucleus through the utilization of operating system processes. The allocation of a child process address space reflects the control hierarchy of its parent. The scale notwithstanding, process scheduling takes place without considering it [13]. The scheduling algorithm employed by StarOS is based on priority and is preemptive. Assigning priority levels to processes is a widely adopted practice in which processes of greater priority are accorded precedence over those of lower priority during execution [14]. The CPU is relinquished to the process with the highest priority, that is, in a ready state. StarOS offers several mechanisms to prevent priority inversion, guaranteeing that those with lower priority do not impede processes with higher priority. Priority inheritance is a mechanism that temporarily elevates the importance of a lower-priority process to match that of a higher-priority process when the former possesses a resource that the latter requires. The tool above serves to avert the occurrence of priority inversion, a situation in which a function with a higher priority is impeded by a circle with a lower focus with a resource that the higher-priority process necessitates. The process of CPU scheduling involves assigning priorities to individual threads. Prioritization is a fundamental aspect of process and thread management, whereby each function is assigned a priority, and each line is allocated a relative importance within its corresponding process [15]. The summation of the significance of a circle and the relative priority of a thread determines the absolute priority of the line. The kernel records the importance assigned to every line in the ACTIVE state and executes the line with the most excellent total focus. In a multiprocessor system comprising k CPUs, the k threads with the highest priority are implemented.

The QNX process-scheduling primitives adhere to the specifications outlined in the draft version of the POSIX 1003.4 (real-time) standard [16]. QNX offers a comprehensive preemptive mechanism facilitating prioritized context switching through round-robin, FIFO, and adaptive scheduling. Upon the emergence of the POSIX 1003.4 standard from its draft status, the microkernel and system processes will undergo evolution to align with the standard. Each software application has a distinct package with an execution model, constructs, and scheduler. Conversely, SPIN solely outlines rudimentary execution contexts, known as strands. It should be noted that applications can only schedule the execution time of application threads, not that of the kernel. A comprehensive scheduler at the user level can be established in SPIN using a spindle per program, simulating scheduler activations [17]. Upon each context switch at the kernel level, the spindle transmits a notification to the thread's address space, indicating the alteration in its scheduling status. A library at the user level is responsible for implementing thread management primitives that are specific to the application. The exokernel architecture delegates the responsibility of thread management and scheduling to applications. The exokernel architecture generates and administers threads without executing thread scheduling operations internally. The burden of determining the following line to be executed lies with the application. The approach towards process scheduling offers considerable adaptability, given that applications can devise their scheduling algorithms to their requirements [19]. Nonetheless, meticulous oversight by the application developer is imperative to guarantee that threads are scheduled suitably.

The L4 microkernel architecture employs a priority-based round-robin methodology to manage process scheduling [20]. The utilization of TCBs and the implementation of lazy scheduling were used. The task frequently transitioned a thread's state between the blocked and runnable states. This resulted in inherent pathological timing behavior. The Benno scheduling has replaced the scheduling above approach in newer versions of L4. The Mach operating system employs preemptive scheduling. In the user mode, threads with higher priority are given precedence to execute on the processor over those with lower priority. Similarly, in the kernel mode, real-time threads with higher priority are granted permission to run before those with lower priority Fiasco.OC microkernel is responsible for managing process scheduling in the L4re microkernel. The system employs a real-time scheduling algorithm that combines fixed focus and round-robin scheduling [21]. The seL4 microkernel employs a preemptive round-robin scheduler to accomplish process scheduling [22]. There are 256 distinct priority levels, and each thread possesses a maximum controlled priority (MCP). Threads are endowed with a practical focus, which functions as their priority. If a thread capability is provided, threads can modify the importance of another thread if the MCP is utilized.

The NOVA micro-hypervisor incorporates a preemptive priority-driven round-robin scheduler and employs a solitary run queue for each CPU [23]. The factor above exerts an impact on the decision-making process of the micro-hypervisor about dispatch. In the dispatch process, an execution context may persist until the depletion of its scheduling context's time quantum or until the release of a scheduling context with higher priority. Upon invocation, the scheduler selects the scheduling context with the highest priority from the run queue. Subsequently, the task above is sent out for execution, and its corresponding context is merged. The scheduling methodology employed by the Muen Separation Kernel is a round-robin approach based on fixed preferences [24]. Additionally, the Muen Separation Kernel uses virtualization techniques to classify the system into multiple entities. The system employs traps to manage these subjects' scheduling and process external interrupts.

B. Memory Management

The RC 4000 Nucleus employs a flat memory model, whereby the entire memory is regarded as a unified address space. The address space is partitioned into three regions by the operating system. The kernel space refers to the portion of a computer's memory reserved for the operating system's core functions and services. The area of memory utilized by the operating system kernel is called the region of memory. The contents of this entity encompass both the code and data structures used by the kernel to manage the various resources of the system effectively. Academic: The user space refers to the specific area of memory allocated for use by processes initiated by the user. Each process possesses a distinct address space that is segregated from other methods. The I/O space pertains to the memory area allocated for the utilization of input/output devices. Access to it is facilitated through specialized instructions or memory-mapped input/output. The RC 4000 Nucleus can reduce dynamic memory allocation using a pool memory scheme [13]. Memory pools refer to predetermined blocks of memory that are available for dynamic memory allocation by processes. The nucleus assists categorized entities, functionalities, and non-blocking communication. The process of mapping an object's data portion to memory enables its access through the standard memory access instructions of the processor. The things in StarOS resemble those found in object-oriented programming languages in that they are exclusively accessible through member functions. A specialized module that allocates objects in memory is called the object manager [14]. Each processor cluster is equipped with one such module. The Chorus platform facilitates the utilization of paged distributed shared memory, following the IVY model [15]. The field of Information Technology employs a dynamic decentralized algorithm wherein distinct managers are responsible for monitoring various pages, and the manager assigned to a particular page changes as the page traverses the system. The fundamental element of data exchange among multiple machines is a segment. Segments are divided into fragments consisting of one or multiple pages. At any given moment, every fragment is either in a read-only state, which may exist on multiple machines, or in a read/write state exclusive to a single machine.

QNX utilizes a hybrid memory management approach involving physical and virtual memory. The physical memory is partitioned into pages that are commonly sized at 4KB. Managing virtual memory is executed through a demand-paging methodology [16], whereby pages are solely loaded into memory upon their necessity. The QNX operating system employs a microkernel architecture, wherein the kernel is designed to be compact and exclusively offers essential services such as process scheduling, memory management, and IPC. Additional functionalities, such as file systems, device drivers, and networking, are executed as distinct processes that operate in the user space. The SPIN memory management interface comprises three fundamental elements, namely physical storage, naming, and translation, which are utilized to break down memory services. These entities are related to crucial memory resources processors provide, namely physical addresses, virtual addresses, and translations. As mentioned, application-specific services initiate communication with this trio to delineate more sophisticated virtual memory notions, including address spaces. The memory system comprises three fundamental components, each facilitated by an individual service interface [17]. The physical address service governs the management of physical pages. The virtual address service facilitates the introduction of capabilities for virtual addresses. It consists of three distinct components: a virtual address, a length, and an address space identifier. These components interact to maintain individuality. The translation service clarifies the relationship between virtual addresses and physical memory. The Exokernel architecture contains a secure binding mechanism that facilitates the safe access of system resources by applications, such as memory pages or Translation Lookaside Buffer (TLB) entries [19]. The secure binding of an application is verified solely during the binding process, thereby enabling efficient authorization of subsequent resource accesses. The hardware is responsible for performing this action in the context of a memory page. An alternative method for transmitting these semantics involves utilizing application code, such as a packet filter, which is authenticated during its submission by the application.

The memory management in the initial L4 microkernel was executed within the user space through a designated memory manager. The grant, map, and flush primitives of the microkernel were employed to utilize pagers. The pagers were engaged in the implementation of the conventional paged virtual memory. The management of address space was carried out using solely low-level mechanisms. The L4 memory management model facilitates allocating significant memory resources by the kernel to conflicting processes via the recursive mapping of a standard frame to distinct pages within their respective address spaces [20]. The security above implication could result in a potential denial of service, which can be prevented through Inter-Process Communication (IPC) management. The seL4 microkernel espouses an extreme perspective on separating policy and mechanism. This pertains to transferring responsibility for memory management to lower elite levels. The seL4 kernel's method of managing resources is distinct in refraining from allocating memory after the boot-up process [22]. The microkernel architecture lacks a heap and instead employs a strictly bounded stack. The memory that becomes available after the kernel's boot-up process is allocated to the initial user-mode process. This memory space is classified as "Untyped" and remains unused. The provision of additional memory spaces required by the kernel is the responsibility of the user-mode process. The components above encompass storage for object metadata, page tables, TCBs, and capability storage. One instance of this scenario occurs when a given process seeks to generate a new thread. In addition to allocating memory for the stack the line will employ, the process must also allocate kernel memory to accommodate the storage of the thread's TCB. The process involves transferring data from untyped memory to the TCB kernel object type through a process commonly referred to as "re-typing." The system's current state allows userland to possess a capability to a kernel object. However, direct access to its data is restricted. The authentication token serves as the means to execute system calls on the thing or initiate the object's destruction, thereby restoring the original Untyped memory. The significance of this memory management is substantial, as it facilitates the verifications that constitute the isolation enforcement of seL4. Moreover, the storage of kernel metadata in memory relies on the memory allocated to the kernel by userland, resulting in a partitioning equivalent to that of userland.

As per the findings of [23], the performance of the NOVA micro-hypervisor is impacted by the reduction of time slices. However, the distinctive methodology employed by NOVA instills optimism for utilizing task-based systems that necessitate minimal time slices. The primary objective of NOVA is to prioritize full virtualization and minimize the distinction between traditional microkernels and hypervisors. The Vancouver VMM operates within the user space of the microkernel architecture and functions without any privileged access. Limited findings were obtained regarding the approach to memory management within the NOVA microkernel. As a separation kernel, the primary kernel utilized in Muen Separation is designed to enforce strict isolation between different software components [24]. The page tables generated by the policy tool are not accessible by the seed. This suggests that the primary kernel is not responsible for managing memory. Nevertheless, it is worth noting that various grains in the separation possess unique stack pages designated for storing per-CPU data on specific pages. The transparency of seeds is attributed to their identical virtual stacks and global storage address values.

C. Inter-Process Communication

The nucleus facilitates inter-process communication through the utilization of message queues. The procedure initiates an asynchronous operation for sending a message. The message transmission is enqueued until the recipient process executes the wait message operation. The design for the wait message is synchronous, indicating that the process will be blocked until a message is received. Following this, the receiver can perform the send response function asynchronously, transmitting the reply to the sender. The sender, in turn, is required to perform the wait-answer operation in a blocking manner to receive the response. The Nucleus IPC system provides a safer inter-process synchronization approach than Dijkstra's semaphores [13]. The nucleus supports typed objects, capabilities, and asynchronous communications. IPC methods, including pipes, sockets, shared memory, message queues, and remote procedure calls (RPC) are often employed in StarOS [14]. In the event of a processor or process failure, it is only necessary to restart the process, which may be done on an alternative processor if necessary. In the event of a process failure, the process is halted. A message containing a capability to the process is dispatched to a designated bailout mailbox assigned explicitly for the failing operation. The system facilitates process fault handling. An actor comprises an address space, a set of resources, and a sequence of threads. A resource that can be utilized is a port, which serves as a buffer for incoming messages to the owner and as the mailbox's address to the sender. It is a fact that a single actor has exclusive ownership of a port. Ports have the potential to undergo migration or aggregation into port groups, thereby facilitating the alteration of the virtual port that receives the message. Message-passing and Remote Procedure Calls (RPC) are the two types of IPC that CHORUS supports. Although message passing provides a service model based on best effort, RPC ensures delivery that occurs precisely once. If the RPC target is unavailable, the caller is appropriately notified. Threads within the same actor have access to a third IPC, which involves exchanging information through their shared memory [15].

The IPC mechanism employed by QNX is characterized by its asynchronous nature and reliance on message passing [16]. Sending a message through a process result in a blocking state until the recipient acknowledges receipt of the notice and responds accordingly. The author posits that while blocking may impose limitations on the functionality of the IPC system, it is a straightforward and expeditious method that can facilitate the implementation of more intricate IPC mechanisms atop this framework. The SPIN tool employs an IPC mechanism based on message-passing, enabling processes to exchange messages [17]. Messages may comprise either data or instructions and are transmitted and received via message queues. The operating system manages the message queue for each process. The SPIN tool facilitates both synchronous and asynchronous message passing. In computer science, synchronous message passing refers to a communication mechanism that halts the sender's execution until the recipient has received the message. On the other hand, asynchronous message passing is a communication mechanism that does not impede the sender's execution. This facilitates inter-process communication in a non-blocking manner, enabling real-time exchange of information between processes. The Exokernel enables a secure transfer of control by directing it to a designated location within a distinct process, achieving the time slice to that process, and establishing the requisite context. The transfer of control does not affect the registers that are perceptible to the application, and these registers can be employed to facilitate the exchange of data among diverse processes. The result is a foundation that is highly efficient and versatile for applications in IPC [19].

The initial L4 implementations were designed to be non-preemptive and did not necessitate concurrency control due to kernel simplification [20]. Consequently, there was an improvement in overall performance. Nevertheless, certain L4 implementations retained preemption points on operations that endured for a prolonged duration. Thus, interrupts were briefly enabled. In the initial iteration of the L4 microkernel, IPC was implemented synchronously. The mechanism above served as the sole means of communication, synchronization, and signaling. Its implementation involved circumventing kernel buffering and all related expenses. L4's selection of lazy scheduling as its preferred scheduling approach was apparent. Moreover, the system employed a user-controlled context switch as its IPC mechanism. The diverse instantiations of IPC exhibited the disadvantage of relying on specific devices. The IPC architecture was abundant in meaning, and as delineated in the literature, "lengthy messages" were employed in POSIX-style Input/Output (I/O) operations toward servers. The utilization of lengthy IPC was found to need to be more compliant with the fundamental principle of minimalism that Liedtke established. The seL4 microkernel offers an IPC mechanism that utilizes message-passing [22]. This facilitates communication among different threads. Kernel-provided services use the mechanism for interactions. The transmission of messages occurs through activating a capability towards a kernel entity. Messages directed toward Endpoints are distinct from those intended for objects, as the kernel only processes the latter. The seL4 microkernel employs endpoints to facilitate IPC.

The NOVA micro-hypervisor facilitates communication between virtual machine monitors (VMM) and enlightened guest Operating Systems through a library of hyper-cars. As per the source, a dedicated address space is allocated for every protection domain. The NOVA micro-hypervisor is utilized to establish a secure virtualization framework that necessitates a decrease in the size of the trusted computing base [23]. In addition, it is noteworthy that the NOVA micro-hypervisor can function as a fundamental microkernel for the Genode microkernel. The topic of interest is Muen Separation Kernel's endeavors to minimize the use of synchronization mechanisms due to their propensity for introducing errors. Nevertheless, it employs two technologies to accomplish this objective in specific areas of its code that necessitate synchronization [24].

TABLE I
SUMMARY OF THE COMPONENTS OF MICROKERNEL

Features	**Process Scheduling**	**Memory Management**	**Inter-Process Communication**
RC 4000 Nucleus	Disk Scheduling	Partitioned	Message Queues
StarOS	Priority-Based Round-Robin, FIFO	Partitioned	Pipes, Sockets, Message Queues, Shared Memory, Remote Procedure Calls
CHORUS	Priority-Based Scheduling	Segmented	Message-Passing, Remote Procedure Calls
QNX	Priority-Based Round-Robin, FIFO	Pagers	Asynchronous Message-Passing, Sockets, Signals, Shared Memory, Remote Procedure Calls
SPIN	Priority-Based Round-Robin	Segmented	Pipes, Semaphores, Message-Passing, Shared Memory
Exokernel	Disk Scheduling	Pagers	Message-Passing, Shared Memory
L4	Priority-Based Round-Robin	Pagers	Message-Passing
L4Re	Priority-Based Round-Robin	Pagers	Synchronous Message-Passing
seL4	Priority-Based Round-Robin	Page Tables	Message-Passing
NOVA	Priority-Based Round-Robin	Page Tables	Asynchronous Message-Passing
Muen	Priority-Based Round-Robin	Page Tables	Message-Passing

V. CONCLUSION

In this study, different microkernel-based operating systems were introduced. The study's starting point was Hansen's RC 4000 Nucleus, driven by the security benefits inherent in a microkernel architecture and the desirability of a system that could be expanded. The reason for implementing the StarOS microkernel methodology was prompted by a specific set of requirements, chiefly those linked to a decentralized environment with limited memory capacity. The CHORUS system made enhancements to certain aspects of StarOS, specifically by providing greater clarity regarding the differentiation between address space and execution within said address space, which is achieved through the implementation of actors and threads. The L4Linux system and QNX operating system are examples of microkernel-based designs that offer real-time guarantees, with the latter being utilized in commercial settings. The approach adopted by SPIN has been founded on the acknowledgment that the microkernel principle of service segregation magnifies the expense of traversing safeguarding domains. The Exokernel operates by securely exporting the hardware without any additional value provided. L4Re is an operating system based on a microkernel architecture and offers a modular design while accommodating various programming languages. The NOVA Microhypervisor is an operating system that utilizes hypervisor technology to provide robust access control and isolation within virtualized environments. The Muen Separation Kernel is an operating system that uses a microkernel architecture to offer strong isolation and access control in environments that are limited in resources, such as embedded systems. The seL4 operating system is explicitly microkernel-based and developed for deployment in high-security settings. Its primary emphasis is on the provision of provable security and formal verification. Over time, there have been advancements in the approach to microkernel design. Nevertheless, it was noted that certain fundamental principles were upheld. These principles have guided the development and design of microkernels. Recent observations indicate that contemporary microkernel designs are primarily iterations of the design concepts followed since the second generation of microkernels, including L4, even when such microkernels are developed from scratch. Because of the categories utilized in the analysis conducted during this research, further research can be conducted on the impact of the general design principles on the security and efficiency of the microkernels.

Acknowledgment

We would like to sincerely thank Dr. Afroza Nahar for her invaluable guidance and support during our research. Her expertise and insights have been instrumental in shaping our understanding of the topic.

References

[1] J. Liedtke, ”Toward Real Microkernels,” Commun. ACM, vol. 39, no. 9, p. 70–77, 1 September 1996.

[2] B. Liu, C. Wu and H. Guo, ”A Survey of Operating System Microkernel,” in 2021 International Conference on Intelligent Computing, Automation and Applications (ICAA), Nanjing, 2021.

[3] K. Levchenko, ”A Survey of Microkernel Operating Systems,” University of California San Diego.

[4] A. S. Tanenbaum, ”A comparison of three microkernels,” The Journal of Supercomputing, vol. 9, no. 1, pp. 7-22, 1 March 1995.

[5] K. Elphinstone and G. Heiser, ”From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?,” in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Farminton, 2013.

[6] O.-A. Isaac, K. Okokpujie, H. Akinwumi, J. Juwe, H. Otunuya and O. Alagbe, ”An Overview of Microkernel Based Operating Systems,” IOP Conference Series: Materials Science and Engineering, vol. 1107, no. 1, pp. 12-52, April 2021.

[7] R. Matarneh, ”Multi Microkernel Operating Systems for Multi-Core Processors,” Journal of Computer Science, vol. 5, no. 7, pp. 493-500, 31 July 2009.

[8] C. Tian, D. Waddington and J. Kuang, ”A Scalable Physical Memory Allocation Scheme for L4 Microkernel,” in 2012 IEEE 36th Annual Computer Software and Applications Conference, Izmir, 2012.

[9] D. Mishra and P. Kulkarni, ”A survey of memory management techniques in virtualized systems,” Computer Science Review, vol. 29, pp. 56-73, August 2018.

[10] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, 10th ed., Wiley, 2018.

[11] ”Category:Microkernels,” 29 March 2013. [Online]. Available: https://en.wikipedia.org/wiki/Category:Microkernels. [Accessed 3 April 2023].

[12] ”microkernel.info,” 12 March 2016. [Online]. Available: http://www.microkernel.info/. [Accessed 3 April 2023].

[13] P. B. Hansen, ”The Nucleus of a Multiprogramming System,” Commun. ACM, vol. 13, no. 4, p. 238–241, 1 April 1970.

[14] A. K. Jones, R. J. Chansler, I. Durham, K. Schwans and S. R. Vegdahl, ”StarOS, a Multiprocessor Operating System for the Support of Task Forces,” in Proceedings of the Seventh ACM Symposium on Operating Systems Principles, Pacific Grove, 1979.

[15] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser, ”CHORUS Distributed Operating Systems,” Computing Systems, vol. 1, no. 4, pp. 305-370, 1988.

[16] D. Hildebrand, ”An Architectural Overview of QNX,” in Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures, Berkeley, 1992.

[17] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage and E. G. Sirer, ”SPIN - An Extensible Microkernel for Application-specific Operating System Services,” University of Washington, Seattle, 1994.

[18] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr and R. Sanzi, ”Mach: a foundation for open systems (operating systems),” in Proceedings of the Second Workshop on Workstation Operating Systems, Pacific Grove, 1989.

[19] D. R. Engler, M. F. Kaashoek and J. O’Toole, ”Exokernel: An Operating System Architecture for Application-Level Resource Management,” SIGOPS Oper. Syst. Rev., vol. 29, no. 5, p. 251–266, 3 December 1995.

[20] A. Au and G. Heiser, ”L4 User Manual, Version 1.14,” University of New South Wales, Sydney, 1999.

[21] ”L4Re Operating System Framework,” [Online]. Available: https://www.kernkonzept.com/l4re-operating-system-framework. [Accessed 14 April 2023].

[22] T. S. Team, ”seL4 Reference Manual, Version 12.1.0,” Data61, 2021.

[23] U. Steinberg and B. Kauer, ”NOVA: A Microhypervisor-Based Secure Virtualization Architecture,” in Proceedings of the 5th European Conference on Computer Systems, Paris, 2010.

[24] R. Buerk and A.-K. Rueegsegge, ”Muen - An x86/64 Separation Kernel for High Assurance,” 2013.