An Overview of Microkernel Based Operating Systems

An Overview of Microkernel Based Operating Systems

Isaac Odun-Ayo[1*], Kennedy Okokpujie[2], Hannah Akinwumi[1], Jesse Juwe[1], Henry Otunuya[1], and Oladapo Alagbe[1]

[1]Department of Computer and Information Sciences, Covenant University, Ota, Ogun State, Nigeria
[2]Department of Electrical and Information Information Engineering, Covenant University, Ogun State, Nigeria
Corresponding Author; isaac.odun-ayo@covenantuniversity.edu.ng +2348028829456

Abstract-
The creation of Operating Systems (OSs) with Microkernels was in response to the various challenges presented by Operating Systems with Monolithic kernels. Microkernel based Operating Systems provides security and flexibility in the system. This paper reviews seven different microkernel-based Operating Systems: L4, GNU Hurd, Genode, L4re, NOVA, seL4, and Muen Separation Kernel. This analysis provides an understanding of the various trends in the Microkernel Based Operating Systems design. Research papers and official documentation of the individual microkernels served as data sources. The result in this paper shows that there has not been a significant variation in the underlying principle of minimality and how Microkernel Operating Systems approach the implementations of their inter-process communication (IPC), memory management, and scheduling.

Key words: Operating system, microkernel, monolithic kernel, Genode, GNU Hurd, Mach, seL4, L4re, NOVA

1. Introduction

Operating Systems are required by Central Processing Units (CPUs) to manage their resources and abstract hardware from applications that execute on them and also the users that interface with such programs. Process lifecycle management, right from their creation up until their termination, and everything between is one of the many functions of the operating system [1]. Operating Systems provide the platform for application programs to be run, known as the user-space. They also link the computer user and the hardware that makes up the computer together, thereby abstracting the hardware from the user [2]. Operating Systems are responsible for the management of memory units, controlling the access of processes to memory, handling communication between the processes, and the overall management of input/output operations through peripheral devices [3]. The Operating System (OS) is a group of system software applications that manage the computer's hardware, and also makes provision for hardware resources that are utilised by programs. It comprises of the kernel space, which is a privileged mode, and the user-space, which is an unprivileged mode. This setup enables a separation between processes to exist [4]. Specifically, the Operating System abstracts the complexity of the hardware, manages the allocation and deallocation of computational resources, and provides isolation and protection. The Operating System runs in kernel mode, where it can control all the hardware and also run any instructions that the machine's hardware has the capacity and capability to execute. Every other software runs in user mode [5]. The functions of a kernel include memory management, security, and stability, I/O communication, resource management, and process management, to mention a few [6]. There are two major approaches to the kernel, namely the monolithic kernel and microkernel [7], as is depicted in Fig. 1.

Early generations of operating systems utilised the monolithic kernel architectures as their form of implementation. All the functions of the operating system were bundled together to make up a single kernel in the monolithic architecture. The bundling of operating system functions led to lots of reliability and security issues, such as bugs in the software. An example of a reliability problem is when bugs in a single software lead to a buffer overrun, thereby causing the entire system to crash. A significant security problem is when skilled malware creators and threat actors take such an error and use it to take over the computer completely[8].

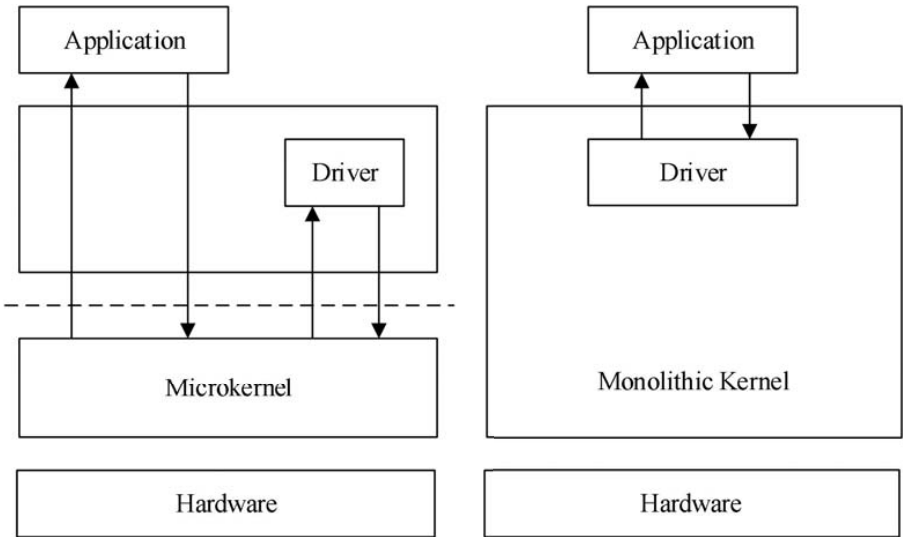

Fig 1. Microkernel Architecture vs Monolithic Kernel Architecture

The problems that arose from monolithic kernels led to the creation of Microkernels. Microkernels manage all the system resources. This paper aims to discuss key concepts of microkernels, alongside their issues and developments. This paper examines various generations of microkernels. The rest of the paper is as follows: section 2 is an overview of related literature contributed by other researchers. Section 3 discusses various implementations of microkernels, while Section 4 examines trends in microkernel design. Section 5 concludes and suggests future areas of research.

2. Related Work

Three microkernels were evaluated [1]. The microkernels were Amoeba, Mach, and Chorus. The evaluation included comparisons and contrasts in the areas of microkernel memory management, process management, and communication. The paper observed that the three microkernels have similar implementation though they were designed independently by different groups.

[9] examines the development of the L4 microkernel over 20 years. A comparison was made between the design specification implemented in the early version L4 by its originator Liedtke and the modern-day L4 microkernel. Microkernel design principles of the L4 were discussed: minimality, IPC (Interprocess Communication), user-level design drivers, and resource management. The paper also outlined the design decisions and implementation of the IPC, which

makes it faster. These include the implementation language, non-standard calling convention, non-portability, direct process switch, preemption, lazy scheduling, and strict process orientation and virtual TCB (thread control block) array. The lessons learnt in those 20 years were examined in [10]. In [11], a model for multicore operating systems was proposed, which is a multi-microkernel approach to system processing as opposed to the single-microkernel approach. A multi-microkernel operating system is divided into two: a master and slaves. The master microkernel will be responsible for the vital services in the system. At the same time, the slaves will be responsible for other minor operations in the system, like establishing communication between two processes.

[12] extensively talks about the Mach microkernel. The key features of the microkernel were discussed: task and thread management, interprocess communication, memory object management, system call redirection, device support, user multiprocessing support and multicomputer support. Also, the paper compared three versions of the Mach microkernel: Mach 2.5, Mach 3.0 and Mach 3.0 Speedup, and two others: SunOS4.1 and Ultrix 4.0 using some test criteria like filesystem, compilation, read (cached), read (uncached), write() and getpid. [13] identified one of the primary functions of an Operating System as memory management. This function is of what kernel architecture type agnostic. Physical Memory Allocation (PMA) requests are usually made upon boot-up of an Operating System. These PMA requests are dynamically made by programs running within the OS. Less than 8% of the physical memory is usually reserved and managed by the kernel-mode PMA, while the user mode Physical Memory Allocator (PMA) the rest. According to [14], memory is divided into independently managed chunks, and these are assigned to processes and threads at runtime of the system.

According to [15], there are two main types of interprocess communications: Shared memory: a portion of memory is created, and then distributed among cooperating processes. Communication then occurs between these processes by them reading and writing to the shared memory space, individually. This is best suited for the exchange of extensive data and is also much faster than message passing as it requires fewer system calls. Message passing: for this to occur, the processes exchange messages directly with each other. This works well with little quantities of data, as there is no need to avoid conflicts because the processes are talking directly with each other, while not sharing the same address space. This has the most significant benefits in distributed computing environments.

[16] talks about the GNU Hurd's design and how it was motivated by a desire to fix some observed problems in Unix. For the most part, several rules that restrict users remain as relics of the design and implementation of the system's mechanisms. It also talks about how the GNU Hurd adopts an object-based architecture and defines interfaces in order to increase integration and extensibility. This paper is, first of all, a description of the Hurd's design objectives and a summary of its architecture, mainly because it represents a deviation from Unix's. It further goes on to evaluate Mach, the microkernel on which the Hurd is designed, highlighting the design constraints imposed by Mach as well as a range of limitations that its architecture poses for multi-server systems.

[17] talks about the flexibility of the microkernel infrastructure operating system. Much of the study was a contrast between FMI/OS and other OS architectures in order to answer some questions like why one should compete with UNIX, why there's any need to compete with systems that already have established success and placement in the market. This paper also looks into the inner-workings of other OS's to make it easier to understand how operating systems work. This paper provides a basic understanding of how operating systems behave, how the FMI/OS operates, and how it addresses and resolves the many problems identified in past monolithic and layered kernels.

[18] studies the scalability of Fiasco.OC, a state-of-the-art microkernel implementation. A new personality, OmniRE, aimed at being multicore scalable is presented. Compared to L4Re, OmniRE seeks to reduce contention by decentralising resource management, scheduling and access to the kernel. The design also aimed to reduce inter-process communication (IPC) across CPUs by localising resource capabilities such as page-fault handling. Experiments were conducted to compare OmniRE against L4Re as well as Linux on a 48-core AMD server and a 6-core Intel workstation. The results suggested that OmniRE offers better scalability than L4Re and can actually surpass the absolute efficiency of Linux in-memory page management at higher core counts.

[19] presents a design of a separation kernel for the Intel x86 architecture using the latest Intel hardware features in his thesis. The open-source prototype written in SPARK shows the feasibility of the intended design and the incorporation of SPARK's proof capability improves the confidence of the accuracy of the implementation. [20] developed and implemented a virtualisation architecture that can host several unmodified host operating systems. Its reliable computing base is at least an order of magnitude smaller than that of existing systems. In addition, their implementation of recent hardware outperformed contemporary full virtualisation environments.

[21] discusses the improvements in the processor architecture to integrate the functions of the microkernel to improve the performance of task-based systems. Part of the CPU overhead is caused by the microkernel running the scheduling algorithm and the context switching. The findings of the experiments carried out show that using this approach, the output is entirely independent of the time-frame, while the traditional approach (software implementation) is decreased by 79 per cent as the time-frame decreases. [22] presents a solution that uses virtualisation to host a slightly modified Linux on the Fiasco microkernel. This solution shows a significant improvement in performance compared to previous solutions, supports SMP, is well integrated and has a significantly reduced resource footprint. It also illustrates how the para-virtualised kernel can be used to achieve true virtualisation. [23], [24] presents in-depth coverage of the detailed, machine-checked formal verification of seL4, a general-purpose operating system microkernel. It discusses the design of the kernel used to make its verification workable. It describes the functional correctness proof of the kernel C implementation and also includes further steps that transform their experimental results into a comprehensive formal kernel verification.

Shropshire in [7] examines the differences in how microkernel and monolithic architectures implement virtualisation. He states that microkernels utilise hypercalls which are paravirtualisation concepts wherein the virtual machine kernel is altered. These hypercalls utilise functions that are found within the kernel to bridge user-mode applications. This is similar to what is obtainable from system calls. Specific subsystems are sent deliberate system calls by the microkernel hypervisors. Interrupts are used by microkernel hypervisors to communicate with virtual machines. These interrupts determine which events require immediate attention.

3. Materials and Methods

3.1. Overview of microkernel

This review paper made a selection of microkernels for evaluation from [25], to ensure that there is a fair representation of microkernels from research only microkernels all the way to production level microkernels. This review paper was focused on how the individual microkernels addressed three significant features of every microkernel, namely their interprocess communication, process scheduling and memory management. This microkernel is considered to be the grandfather of the recent generations of microkernel design. This is observed in all the variants that exist to the fundamental design that was created by Liedtke in [26]–[28]. Liedtke designed the L4 microkernel as a response to the challenges identified in the first generation of microkernel operating systems as was observed in the Mach microkernel [1], [12], and the approach that he used in the L3 microkernel [14]. This design was a radically different approach to what was done earlier in trying to solve the problems identified in monolithic operating system designs [8], [26], [27]. The approach by Liedtke in [27] by providing a solution to the ideas as stipulated by Hansen in [29] with respect to the simplification of implementation by ensuring that only what is necessary is left within the kernel space, and every other thing is moved to the user-space. L4 was therefore used to supply the abstractions on address space, threads and IPC. L4 ensured that all its device drivers had only user-level privileges [10]. This was based on the principle of minimality that formed the basis of L4's design [27].

Variants of the original L4 microkernel have found their way into multiple use cases such as mobile devices [30] and also in mobile security implementations [31][32].

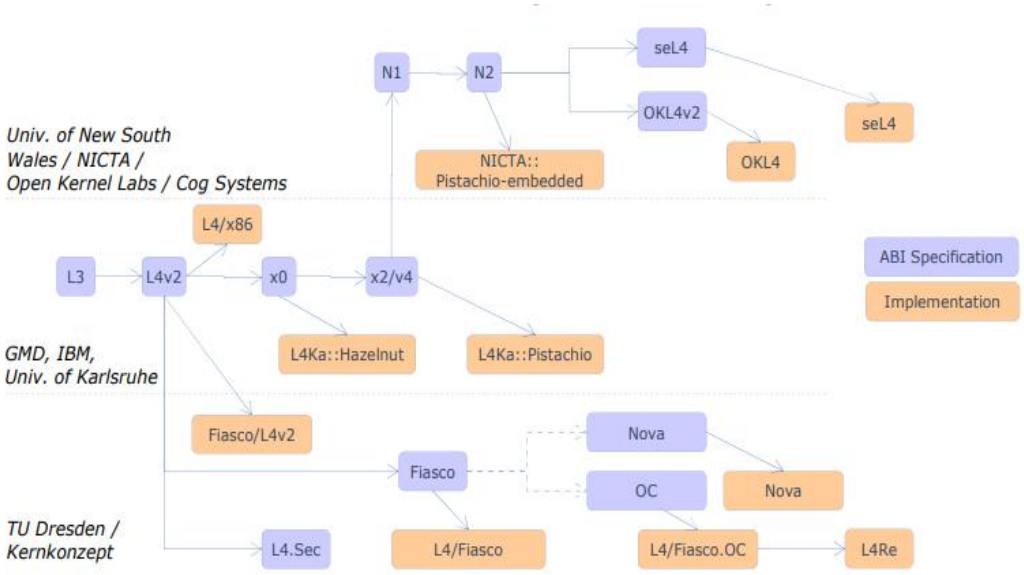

Fig 2. L4 Family of Microkernel Operating Systems

3.1.1. GNU Hurd

GNU Hurd is a UNIX-compatible microkernel operating system. Michael Thomas Bushnell in the year 1991 began the design and implementation of the GNU Hurd. The Hurd, which was one time called Alix, is an operating system which came from a move to have a free operating system on which GNU programs would run [33]. It is a core component of the GNU project [34]. Hurd is a replacement of the UNIX kernel on which GNU programs ran in the past [34] and also stemmed due to some shortcomings [33] and restrictiveness of the UNIX platform [17]. Hurd emphasises both security and flexibility. GNU Hurd is implemented on top of the Mach 3.0 microkernel thereby building on the existing abstractions: task, thread, port, port right, message, memory object and address space [15], [35] - already present in Mach 3.0. GNU Hurd also implements the Mach 3.0 by leveraging on the concept of virtualisation. This is possible due to the object-oriented nature of Mach [33], [35].

3.1.2. Genode

Genode is a free, open-source operating system architecture consisting of a microkernel abstraction layer and a set of user-space components. Genode supports the x86 (32 and 64 bit), ARM (32 bit), and RISC-V (64 bit) CPU architectures. Modern features like IOMMUs and hardware virtualisation can be used on x86 architecture. Furthermore, Genode can utilise the advantages presented by TrustZone and virtualisation technology, on ARM architectures. Genode can be deployed on a variety of different kernels including most members of the L4 family such as NOVA, seL4, Fiasco.OC, OKL4 v2.1, L4/Fiasco, etc. [36]. Every piece of functionality in the Genode architecture should only be seen by those parts of the system which the architecture ultimately depends on while remaining hidden from all other unrelated parts. This reduces the likely surface of attack through individual security functions.

3.1.3. L4re

The L4re microkernel is one of the descendants of the original L4 microkernel. It follows the design philosophy of the L4 family of microkernels such that the only reason why a part of a kernel will be resident within the kernel space, is because moving it out to the user space will prevent the implementation of the system's functionality. The L4 Runtime Environment, as it is called, is an Operating System framework that is used for the creation of real-time systems that also have security, safety and virtualisation requirements [18]. The Runtime Environment consists of the Fiasco.OC microkernel serving as its lowest-level and only highest privileged program. It excludes complex services from within itself and is only populated by kernel objects [37]. The combination of both the Fiasco.OC microkernel and the L4re services provide the complete L4re microkernel that has all necessary services to run user-level applications. The L4re microkernel can scale from embedded systems all the way up to High-Performance Computing (HPC) systems, and as such, has been implemented in security and mobile devices, and also in automobile technology [38].

3.1.4. NOVA Micro-Hypervisor

The name of this micro-hypervisor is a recursive acronym that stands for NOVA OS Virtualization Architecture [19]. NOVA is a combination of the functions of the microkernel and hypervisor. It provides a Trusted Computing Base (TCB) that is small for user applications and also for the virtual machines which run on it. NOVA uses an authorisation model that is capability-based and provides basic mechanisms for spatial and temporal separation. Virtualisation, communication, scheduling, and the management of platform resources are also provided by its TCB. NOVA micro-hypervisor provides a virtualisation architecture that is secure due to its lightweight components that can be designed independently and also developed and verified for correctness [20].

3.1.5. seL4

The seL4 microkernel is a third-generation Operating System microkernel that was designed from entirely for use in security and safety-critical systems. The unique assurance of this microkernel includes formal, machine-checked proofs that the implementation (at the level of the executable binary) is functionally correct against a formal model and that the formal model enforces integrity and confidentiality (ignoring timing channels) [39]. Like other security-oriented systems [40][41], seL4 uses capabilities [42] for access control. There must be an authorisation of access by an appropriate capability.

3.1.6. Muen Separation Kernel

The Muen Separation Kernel is a type of specialised microkernel that has been proven formally, to have no errors at its source code at the time of execution. It is the first of its kind. Furthermore, being a separation kernel, it makes available an environment where multiple components can execute and also interact with each order based on a pre-specified policy. Non-adherence to the policy will imply that such components execute in isolation. This limits the potential of side and covert based channel attacks. Furthermore, the Muen Separation Kernel is generally smaller and more static than dynamic microkernels. This aids in the application of formal verification

techniques [19], [24]. The Muen Separation Kernel has been utilised as the base of highly complex security solutions that are in development [19].

3.2. Process Scheduling

In the L4 microkernel architecture, process scheduling is handled by the use of a priority-based round-robin approach [10]. Thread Control Blocks (TCBs) were utilised, and lazy scheduling [10] was chosen as the form of scheduling implementation. This involved moving a thread's state from runnable to blocked, and vice versa, frequently. This led to inherent pathological timing behaviour. This method of scheduling was dropped for the Benno scheduling in more recent implementations of L4. Mach implements preemptive scheduling. In the user-mode, higher priority threads are assigned to the processor to execute before the lower ones and also in the kernel-mode, higher priority real-time threads are allowed to execute before the lower ones. In the L4re microkernel, process scheduling is handled by the Fiasco.OC microkernel. It features a fixed priority, round-robin real-time scheduling.[38][43].

In NOVA micro-hypervisor, a preemptive priority-driven round-robin scheduler is implemented, alongside a single runqueue per CPU. This influences how dispatch decisions are made by the micro-hypervisor. During dispatch, an execution context can exist until the time quantum of its scheduling context has been depleted, or if a higher-priority scheduling context is released. All of these happen upon invocation, as the scheduler chooses the highest priority scheduling context out of the runqueue. This is then dispatched, and its execution context is joined to it. To achieve process scheduling, seL4 microkernel utilises a preemptive round-robin scheduler. Here there exist 256 different priority levels, and there is a maximum controlled priority (MCP) that is possessed by every thread. Also, threads possess a priority that serves as its effective priority. Threads have access to modify the priority of another thread provided that it has provided a thread capability that the MCP will be used from [44], [45]. The approach of the Muen Separation Kernel to scheduling is a fixed priority based round robin. Furthermore, the Muen Separation Kernel utilises virtualisation to isolate the system into multiple subjects. It also utilises traps to schedule these subjects and also to handle external interrupts [46].

3.3. Memory Management

Memory management in the original L4 microkernel was handled in the user space by a memory manager. Pagers were utilised through the microkernel's grant, map and flush primitives. The pagers were further utilised in the implementation of the traditional paged virtual memory [47]. There were only low-level mechanisms that were utilised in the management of address space [48]. The model adopted in L4 memory management allowed for clashing processes to have the kernel utilise great amounts of memory through the recursive mapping of a similar frame to pages that are different in their individual address spaces. This had the security implication of being a potential source of a denial of service, and could only be avoided via the controlling IPC [48].

A Least Recently Used (LRU) algorithm is used by Mach to decide on which page to keep in the main memory and which to discard. When a page is removed from memory, Mach communicates with the program in the user-space associated with the page through its pager which handles the evicted page. When a page fault occurs during a program's execution, Mach

communicates with the pager again in order to resume program execution. Just like other L4 based microkernel operating systems, memory management here is achieved through the use of a pager. This is done by user-level applications. The way in which the kernel grants memory securely is through a concept known as memory mapping [48], [49]. According to [21], NOVA micro-hypervisor performance is affected by time slice reduction, and the innovative approach of NOVA gives hope to applications of task-based systems which need very short time slices. NOVA considers full virtualisation as its primary aim and reduces the difference between traditional microkernels and hypervisors. It runs the Vancouver VMM, which runs entirely unprivileged in the user space of the microkernel architecture [50]. Not much was discovered in terms of how memory management is addressed in the NOVA microkernel

An extreme view of policy-mechanism separation is taken by the seL4 microkernel [51]. This is the delegation of all memory management to lower privileged level. This approach to kernel resource management by seL4 is unique, as after booting up, the kernel never allocates memory. The microkernel possesses no heap and utilises a stack that is strictly bounded. Any memory that is freed up after the kernel has finished booting is then transferred to the initial user-mode process. This is designated as an "Untyped" (unused) memory space. Other memory spaces that are required by the kernel must be supplied by the user-mode process. These include memory for object metadata, page tables, thread control blocks (TCBs) and capability storage [52]. An example of this is when a process wants to create a new thread; besides the provision of memory for the stack that will be utilised by the thread, also it must also be handed to the kernel memory for storing its TCB. This happens by "re-typing" some "Untyped" memory into the TCB kernel object type. Userland now holds a capability to a kernel object; it cannot access its data directly. This capability is the authentication token that is used to perform system calls on the object or even destroy the object, which leads to the recovery of the initial Untyped memory) [45],[23]. The impact of this memory management is huge, as it enables the proofs that make up seL4's isolation enforcement. Furthermore, as kernel metadata is stored in memory provided to the kernel by userland, it is as partitioned as userland [9], [45], [53], [54]. Being a separation kernel, the main kernel in Muen Separation Kernel does not have access to the page tables that are created by the policy tool. This implies that the main kernel does not deal with memory management. However, other kernels within the separation have distinct stack pages where specific pages where they store per-CPU data. This is fully transparent to the kernels due to their global storage address values and their virtual stacks being the same [24].

3.4. Interprocess Communication

Original L4 implementations were non-preemptible and required no concurrency control due to a simplification of the kernel. This led to better performance generally. However, some L4 implementations still contained some preemption points on operations that ran for a long period of time. This led to interrupts being enabled briefly [30]. On the original version of the L4 microkernel, IPC was of the synchronous type [48]. This was used as the only mechanism for communication, synchronisation and signalling, and was implemented by avoiding buffering in the kernel, alongside all associated costs [48]. This was evident by L4's utilisation of lazy scheduling as its chosen method of scheduling. Furthermore, a user-controlled context switch was utilised as its IPC model. The various implementations of IPC had the drawback of being

device-dependent. This led to the initial notion by [47] in 1995 that microkernel implementations should not strive for portability due to all the overhead costs and specific hardware optimisation opportunities. Furthermore, the IPC structure was rich in semantics, and as specified in [48], "long messages" were utilised in POSIX-like I/O activities to servers. Finally, this use of long IPC was in violation of the overarching philosophy of minimality that Liedtke enshrined [55]. In Mach, Interprocess Communication is done using two of Mach's abstractions: message and port [2]. A port is simply the message queue which tasks can be given rights to access. Only one task can have a 'receive right' on a port with any number of tasks having a 'send right'. Those are the two capabilities a task can possess in Mach - send or receive. This is to ensure security within the system. Interprocess communication in Genode is achieved via two thin abstractions that build upon each other [36]. These include a Physical Memory Allocator (PMA) that is created for each core to achieve good scalability in L4 microkernel. This not only serves to reduce contention, but it also effectively reduces the overhead in IPC that is critical to the performance of the microkernel. [13]

If an aspect of the system gets compromised, this compromise is limited to just the particular aspect of the system that was compromised alongside other parts that it serves as dependencies to. Functionalities that are unrelated remain unaffected. The approach that is utilised by Genode to achieve this security function is the composition of the system by many individual components that ultimately interact with each other. Each of these components serves clear roles and also use specifically defined interfaces in communicating with their peers. [36]. Just like other L4 based systems, Inter-process communication in the L4re microkernel is based off a synchronous form of communication. This implies that there has to always be an active rendezvous between interacting partners for inter-process communication to occur [48]

NOVA micro-hypervisor supports a library of hypercalls that can be used to allow communication between enlightened guest Operating Systems and the Virtual Machine Monitors (VMM). According to [20], a host address space is preserved for each domain of protection. NOVA micro-hypervisor is used to provide a secure virtualisation architecture where a reduction in TCB size is required. Furthermore, NOVA micro-hypervisor can serve the Genode microkernel as one of its base microkernels. A message-passing IPC mechanism is provided by the seL4 microkernel. This is used for interaction between threads. Kernel provided services utilise this same mechanism for interactions. Messages are sent by the invocation of a capability to a kernel object [45]. For messages for other threads are sent to Endpoints, while the messages that are processed by the kernel are those that are intended for objects [45]. To achieve this, seL4 microkernel utilises endpoints which allow for this communication. Muen Separation Kernel tries as much as possible to avoid the utilisation of synchronisation due to its error-prone nature. However, for certain aspects of its code that requires some form of synchronisation, it utilises two technologies in achieving this. These include Spinlock, as recommended by Intel SDM [56], and also the Barrier mechanism [57].

4. Results and Discussions

Source lines of code (SLOC), is a metric that is generally used to measure a software program or its codebase based on its size. It is also known as lines of code (LOC). SLOC is a general

identifier used by adding a number of lines of code used to write a program. A microkernel can be implemented in less than twenty thousand lines of source code due to its limited functionalities. This further helps in its ease of maintenance. All the microkernel operating systems observed maintained the minimality principle as their fundamental guiding principle. This was observed in their ensuring that only the items that were paramount to kernel operations were present in kernel space. Every other element was moved to the user-space. It was observed that there had been a shift in the choice of languages that were utilised in the design of the microkernel operating systems. This is evidence in microkernels like seL4 that dropped C++ for the C language, due to its verification requirements. Formal verification is the process of checking and guaranteeing the correctness of a program. This is done using a list of properties which a program should possess in the form of a formal specification. Programs are viewed in the form of mathematical methods and are evaluated as mathematical proofs [23], [24], [58]

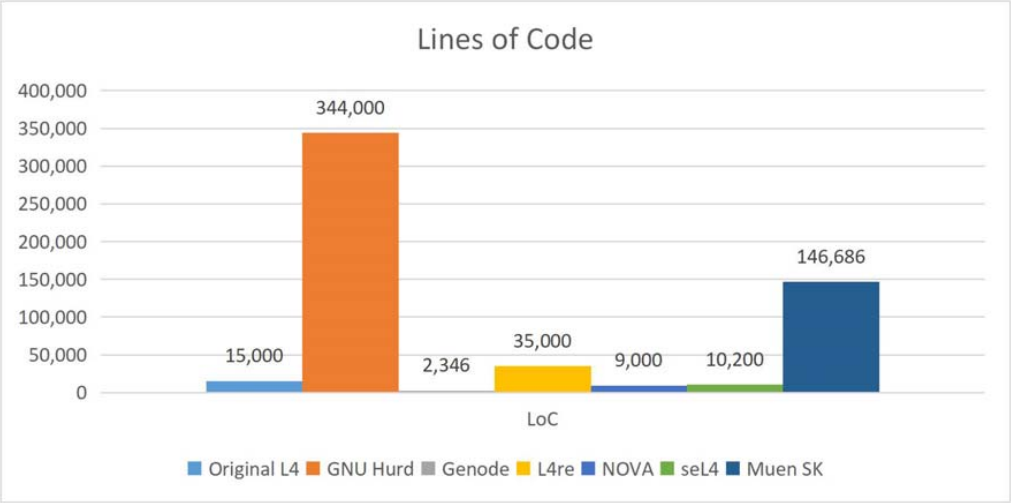

Fig 3. SLOC of Reviewed Microkernel Based Operating Systems

Table 1. Summary of Microkernel Design

Feature / Microkernel	L4	GNU Hurd	Genode	L4re	NOVA	seL4	Muen Separation Kernel
Generation	2nd	2nd	3rd	3rd	3rd	3rd	3rd
Scheduling Type	Priority-Based RR	Priority-Based RR	Priority-Based RR	Priority-Based RR	Priority-Based RR	Priority-Based RR	Priority-Based RR
Memory Management	Pagers	Pagers	Page tables	Pagers	Page tables	Page tables	Page tables
Use type	Active development	Active development	Production	Production	Experimental	Production	Experimental

License Model	BSD	GPLv2+	GPLv3	GPLv2	GPLv2	GPLv2	GPLv3
Non-Preemptible	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Formal Verification	-	-	-	-	-	Yes	Ongoing
Implementation Language	ASM	C and LISP	C++	C++	C++	C and ASM	SPARK
LoC	15,000	344,000	2,346	~35,000	9,000	10,200	146,686
Minimality	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Architecture Supported	Alpha	x86	x86, ARM, RISC-V	x86, ARM, MIPS	x86	ARMv6	x86

5. Conclusion

Over the years, there have been advances in the way microkernel design has been done. However, some fundamental principles were observed to have been maintained. These principles have governed overall microkernel development and design. Furthermore, it has been observed that in recent times, the microkernels that are being designed have been mere iterations of the design concepts adhered to since the second generation of microkernels through the L4, even in instances where such microkernels were built from the ground up. In view of the categories utilised 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.

Acknowledgements

We acknowledge the support and sponsorship provided by Covenant University through the Centre for Research, Innovation, and Discovery (CUCRID). There are no conflicts of interest.

References

[1] A. S. Tanenbaum, "A comparison of three microkernels," *J. Supercomput.*, vol. 9, no. 1–2, pp. 7–22, 1995, doi: 10.1007/BF01245395.

[2] A. SILBERSCHATZ, P. B. GALVIN, and G. Gagne, “Operating Systems Concepts,” 2013.

[3] W. Chengjun, "The Analyses of Operating System Structure," in *2009 Second International Symposium on Knowledge Acquisition and Modeling*, 2009, vol. 2, pp. 354–357, doi: 10.1109/KAM.2009.265.

[4] A. Gautam, A. Kumari, and P. Singh, "The Impact of Architecture on the Performance of Monolithic and Microkernel Operating System," *Int. J. Adv. Res. Comput. Sci. Softw. Eng. Concept Object Recognit.*, vol. 5, no. 3, pp. 107–110, 2015.

[5] A. S. Tanenbaum and H. Bos, *Modern Operating System*, no. 4. 2015.

[6] R. E. (Carnegie M. U. Bryant and D. R. (Carnegie M. U. and I. L. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2011.

[7] J. Shropshire, "Analysis of monolithic and microkernel architectures: Towards secure hypervisor design," *Proc. Annu. Hawaii Int. Conf. Syst. Sci.*, pp. 5008–5017, 2014, doi: 10.1109/HICSS.2014.615.

[8] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?," *Computer (Long. Beach. Calif).*, vol. 39, no. 5, pp. 44–51, 2006, doi: 10.1109/MC.2006.156.

[9] K. Elphinstone and G. Heiser, "From L3 to seL4: What have we learnt in 20 years of L4 microkernels?," *SOSP 2013 - Proc. 24th ACM Symp. Oper. Syst. Princ.*, pp. 133–150, 2013, doi: 10.1145/2517349.2522720.

[10] G. Heiser and K. Elphinstone, "L4 microkernels: The lessons from 20 years of research and deployment," *ACM Trans. Comput. Syst.*, vol. 34, no. 1, 2016, doi: 10.1145/2893177.

[11] R. Matarneh, "Multi microkernel operating systems for multicore processors," *J. Comput. Sci.*, vol. 5, no. 7, pp. 493–500, 2009, doi: 10.3844/jcssp.2009.493.500.

[12] D. L. Black *et al.*, "Microkernel Operating System Architecture and Mach," *J. Inf. Process.*, vol. 14, no. 4, pp. 11–30, 1991.

[13] C. Tian, D. G. Waddington, and J. Kuang, "A Scalable Physical Memory Allocation Scheme for L4 Microkernel," *2012 IEEE 36th Annu. Comput. Softw. Appl. Conf.*, pp. 488–493, 2012.

[14] D. Mishra and P. Kulkarni, "A survey of memory management techniques in virtualised systems," *Computer Science Review*, vol. 29. Elsevier Ireland Ltd, pp. 56–73, 01-Aug-2018, doi: 10.1016/j.cosrev.2018.06.002.

[15] A. SILBERSCHATZ, P. B. GALVIN, and G. Gagne, *Operating System Concepts*, 10th ed. WIley, 2018.

[16] N. Walfield and M. Brinkmann, "A critique of the GNU hurd multi-server operating system," *Oper. Syst. Rev.*, vol. 41, pp. 30–39, 2007, doi: 10.1145/1278901.1278907.

[17] D. Hammond, "FMI/OS: A Comparative Study," 2006.

[18] J. Kuang, D. G. Waddington, and C. Tian, "Towards a scalable microkernel personality for multicore processors," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8097 LNCS, pp. 620–632, 2013, doi: 10.1007/978-3-642-40047-6_62.

[19] R. Buerki and A. Rueegsegger, “Muen - An x86/64 Separation Kernel for High Assurance,” 2013.

[20] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-Based Secure Virtualization Architecture," doi: 10.1145/1755913.1755935.

[21] L. P. Dantas, R. J. De Azevedo, and S. P. Gimenez, "A novel processor architecture with a hardware microkernel to improve the performance of task-based systems," *IEEE Embed.*

Syst. Lett., vol. 11, no. 2, pp. 46–49, 2019, doi: 10.1109/LES.2018.2864094.

[22] S. Liebergeld, "Lightweight Virtualization on Microkernel-based Systems," 2010.

[23] G. Klein *et al.*, "Comprehensive Formal Verification of an OS Microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, 2014, doi: 10.1145/2560537.

[24] I. Haque, D. D's ouza, P. Habeeb, A. Kundu, and G. Babu, "Verification of a Generative Separation Kernel."

[25] "Microkernels - The component-based operating systems." [Online]. Available: http://www.microkernel.info/. [Accessed: 08-Feb-2020].

[26] J. Liedtke, "Towards Real Microkernels," *Commun. ACM*, vol. 39, no. 9, pp. 70–77, 1996.

[27] J. Liedtke, "On Micro-Kernel Construction," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 237–250, Dec. 1995, doi: 10.1145/224057.224075.

[28] J. Liedtke, "1 Rationale 2 Some -Kernel Concepts," *Technology*, pp. 1–14, 1994.

[29] P. B. Hansen, "The Nucleus of a Multiprogramming System," *Commun. ACM*, vol. 13, no. 4, pp. 238–241, Apr. 1970, doi: 10.1145/362258.362278.

[30] C. van Schaik and G. Heiser, "High-Performance Microkernels and Virtualisation on {ARM} and Segmented Architectures," in *International Workshop on Microkernels for Embedded Systems*, 2007.

[31] "Introduction to Apple platform security."

[32] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4Android: a generic operating system framework for secure smartphones," in *SPSM '11*, 2011.

[33] N. H. Walfield and M. Brinkmann, "A critique of the GNU hurd multi-server operating system," *Oper. Syst. Rev.*, vol. 41, no. 4, pp. 30–39, 2007, doi: 10.1145/1278901.1278907.

[34] "System GNU / Hurd on Architecture x86 System GNU / Hurd on Architecture x86," 2006.

[35] M. I. Bushnell, "The HURD: Towards a New Strategy of OS Design," 1994.

[36] N. Feske, "Operating System Framework 17.05," p. 490, 2017.

[37] M. Partheymüller, J. Stecklina, and B. Döbel, "Fiasco.OC on the SCC."

[38] "L4Re Technology," 2013. [Online]. Available: https://www.kernkonzept.com/l4re.html. [Accessed: 31-Jan-2020].

[39] G. Klein *et al.*, "Comprehensive Formal Verification of an OS Microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 1–70, 2014, doi: 10.1145/2560537.

[40] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro, "The KeyKOS Nanokernel Architecture," in *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, 1992, pp. 95–112.

[41] J. S. Shapiro and N. Hardy, "EROS: A Principle-Driven Operating System from the

Ground Up," *IEEE Softw.*, vol. 19, no. 1, pp. 26–33, Jan. 2002, doi: 10.1109/52.976938.

[42] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Commun. ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966, doi: 10.1145/365230.365252.

[43] "The Fiasco microkernel - Status," 2013. [Online]. Available: https://l4re.org/fiasco/features.html. [Accessed: 31-Jan-2020].

[44] "Trustworthy Systems Team, Data61," 2018, doi: 10.1.1.

[45] T. S. Team, "seL4 Reference Manual Version 3.2.0," no. July, 2016.

[46] R. West, Y. E. Li, E. Missimer, and M. Danish, "A Virtualized Separation Kernel for Mixed-Criticality Systems," *ACM Trans. Comput. Syst*, vol. 34, no. 8, 2016, doi: 10.1145/2935748.

[47] J. Liedtke *et al.*, "Achieved IPC performance (still the foundation for extensibility)," no. June, pp. 28–31, 2002, doi: 10.1109/hotos.1997.595177.

[48] S. Humenda, "Rust and Inter-Process Communication (IPC) on L4Re," no. May, 2019.

[49] A. Lackorzynski, "The L4Re Microkernel," no. June, 2018.

[50] S. Liebergeld, "Lightweight Virtualization on Microkernel-based Systems," 2010.

[51] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/Mechanism Separation in Hydra," in *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, 1975, pp. 132–140, doi: 10.1145/800213.806531.

[52] D. Elkaduwe, P. Derrin, and K. Elphinstone, "Kernel Design for Isolation and Assurance of Physical Memory," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, 2008, pp. 35–40, doi: 10.1145/1435458.1435465.

[53] D. E. Devices, "Secure Microkernel for Deeply Embedded Devices Secure Microkernel for Deeply Embedded Devices," *Fosdem 2017*, 2017.

[54] Q. Ge, T. Chothia, Y. Yarom, and G. Heiser, "Time protection: The missing OS abstraction," *Proc. 14th EuroSys Conf. 2019*, vol. 2019, 2019, doi: 10.1145/3302424.3303976.

[55] T. Notes, "Microkernel development : from project to implementation."

[56] "IA-32 Intel ® Architecture Software Developer's Manual Volume 3: System Programming Guide," 1997.

[57] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[58] S. Amin, "Artificial Intelligence and Formal Verification," 2017.