Building Blocks of a Microkernel: Lessons from seL4

Harshit Raj*

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Abstract

In this report, we provide an analysis of the seL4 microkernel, exploring its building blocks and implementation. The seL4 microkernel is designed to minimize the Trusted Computing Base (TCB) and provide a secure system-building base by moving most kernel functionality to non-privileged user mode. This design enables the microkernel to be formally verified, providing an extra level of assurance in its security and reliability. We examine the seL4 kernel codebase and provide an overview of the kernel objects, components, and capabilities. Additionally, we present a design for building a very minimal microkernel. Overall, this report provides a detailed understanding of a microkernel and its building blocks.

Introduction

Kernel is a piece of code that serves as the core of an operating system. It manages the system's resources and provides a layer of abstraction between the hardware and software. The kernel is responsible for handling tasks such as memory management, process scheduling, device drivers, and input/output (I/O) operations. All these operations take place in a privileged mode, also know as kernel mode. On the other hand, programs operate in user mode which doesn't allow them to directly access hardware resources. Instead, they need to utilize the operating system to access these resources via different abstractions and interfaces.

Over the years this privileged code has increased with addition of newer technologies and this is as dangerous as useful. If any malitious access is allowed here that could compromise the entire system. This has occured on many mainstream system. Linux kernel comprises of 20 Million lines of source code, it is estimated that it contains tens of thousands of bugs [1]. This idea is captured by saying that Linux has a large trusted computing base (TCB), which is defined as the subset of the overall system that must be trusted to operate correctly for the system to be secure [2].

The idea behind a microkernel design is to drastically reduce the TCB to have a secure base to build a system. In a well-designed microkernel, such as seL4, it is of the order of ten thousand lines of source code. As seen in Figure 1 the monolithic kernel structure provides every functionality from kernel mode, i.e. privileged mode making the TCB large. On the other hand, the microkernel structure provides only the bare minimum functionality from kernel mode and the rest of the functionality is provided by user mode programs. This makes the TCB small and secure. However the downside of this approach is that it requires a lot of user mode programs to be written to provide the functionality and this incurs a performance penalty.

In this report we will be looking at the building blocks of a microkernel and how they are implemented in seL4. We will also be looking at the a design to build a teching microkernel using gem5 simulator [3]. This report aims to explore the concept of microkernels and their significance in designing secure operating systems. A microkernel design addresses this problem by minimizing the TCB to only include the essential functions of the kernel.

Microkernel Design

The primary objective of Microkernels is to offer a small and secure kernel that is critical for

---

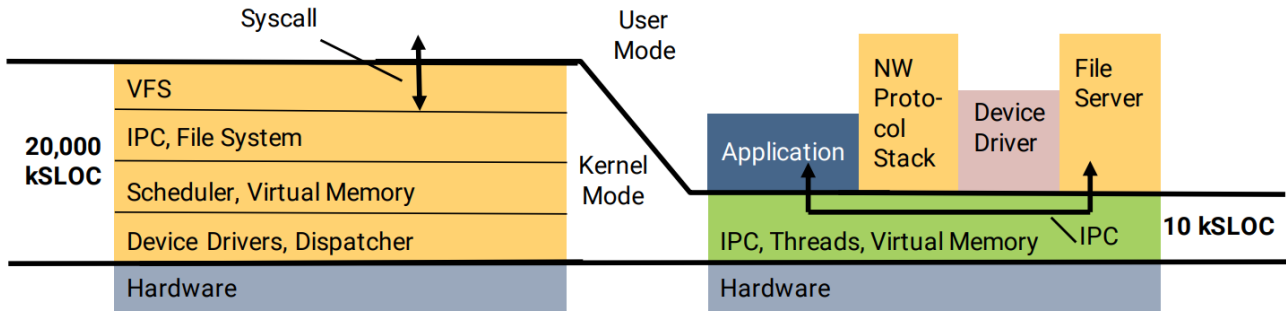*Supervised by: Prof. Debadatta Mishra
Submitted: Apr 25, 2023

Figure 1: *To left is a monolithic structure and microkernel structure on right [2]*

security. The kernel provides only the essential functionality in kernel mode, and the remaining functionality is provided by user mode programs. This approach helps keep the Trusted Computing Base (TCB) small and secure, which reduces the attack surface. A well-designed microkernel, such as seL4, is typically only about ten thousand lines of source code, in contrast to the Linux kernel, which is approximately 20 million lines of code. This significant difference in size results in a much smaller TCB in a microkernel design, as shown in Figure 1. Let us now consider a microkernel design principle.

Minimality & Generality

The principles of minimality and Inter-Process Communication (IPC) performance were the primary drivers of Liedtke's designs. He firmly believed that minimality enhances IPC performance. To express this idea, he introduced the microkernel *minimality principle* as follows: [4]

> *A concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality*

L4 had an implicit driver for the design of kernel mechanisms, which was generality: the aim was to create a foundation on which various systems could be built, almost anything that could run on a processor powerful enough to provide protection.

However, none of the designers of L4 kernels to date claim that they have developed a "pure"

microkernel that strictly adheres to the minimality principle. For instance, all of them have a scheduler in the kernel, which implements a specific scheduling policy, usually a hard-priority round-robin. To date, no one has been able to come up with a general in-kernel scheduler or a viable mechanism that delegates all scheduling policy to the user level without imposing a significant overhead.

In conclusion, microkernels maintain minimality as a crucial design principle and generality as the overall goal. [5]

Building Blocks

In this section, we will briefly examine the various building blocks that make up a microkernel.

Components Unlike traditional operating systems, microkernels do not provide any typical OS services. Instead, microkernel designs use **user-mode** programs called components to provide necessary system services. For example, a file system component provides file system services. These components are loaded into the kernel and run in user mode. The kernel provides the Inter-Process Communication (IPC) mechanism for these components to communicate with each other. Even the **user application** is a component in this design, as depicted in Figure 1.

Kernel Objects Microkernels also include a limited set of well-defined core kernel objects. These objects run in privileged mode and are responsible for providing basic services, including an entry point for interrupt handling, IPC, basic virtual memory management, thread management, and scheduling. These are the only func-

tionalities provided by the kernel. The rest of the services are provided by the components, as shown in Figure 1.

Build System As the components and kernel objects are complex and involved, they need to be configured before the system boots up and sent to appropriate locations in memory. This is the responsibility of the build system, which configures the components and kernel objects and sets metadata for the components.

We will discuss each of these building blocks in more detail later.

Components

Components are user-mode programs that provide system services and also function as application programs. The kernel provides an Inter-Process Communication (IPC) mechanism for these components to communicate with each other, if permitted.

Nature of Components:

- Components are user-mode programs and do not possess special privileges. All logic is implemented in user mode, and components do not have access to kernel mode.
- Components are independent of each other. If one component fails or is compromised, it does not affect the other components. The only impact is on the service provided by the failed component.
- Components communicate with each other using the IPC mechanism provided by the kernel.
- Components must be granted proper access rights at build time to access kernel objects, such as invoking IPC or accessing memory.
- Components are configurable, allowing them to provide different services, interrupt handling, etc., at build time.
- Components are reusable, meaning they can be used in different systems.

Representation of Components

Components are represented as executables using Thread Control Blocks (TCBs). TCBs are data structures that contain information about the component and are used by the kernel to manage the component's execution. TCBs store information needed for context switching, scheduling, priority, IPC buffer, IPC call, capabilities, etc.

Data in TCB The TCB contains various information, including:

- Architecture-specific TCB state, such as the program counter and stack pointer.
- Thread state: a three-word data structure indicating the current state of the component, which can be Inactive, Running, Restart, Blocked on Receive, Blocked on Send, Blocked on Reply, Blocked on Notification, or Idle Thread State.
- Current fault of the component, i.e., the fault that caused the component to be blocked.
- Maximum controlled priority of the component.
- Scheduling context of the component, which contains scheduling parameters such as core and timeslice.
- Scheduling and notification queues.

Capabilities

Capabilities are access rights granted to components to access kernel objects. These access rights are determined at build time and are represented as a data structure in the kernel. Capabilities are configurable at build time and are used to invoke IPC, access memory, etc. Capabilities are essential for implementing the principle of least privilege and determine which component can access which interrupt.

Kernel Objects

Kernel objects are fundamental components of the microkernel that provide essential services such as interrupt handling, inter-process communication (IPC), basic virtual memory management, thread management, and scheduling. These are the only functionalities that the kernel provides directly. All other services are provided by components that incur an IPC overhead. The kernel objects are implemented in privileged

mode. We have already seen one kernel object, the TCB, which is used to manage the execution of components. In this section, we will discuss the other kernel objects, namely system calls, IPC, virtual memory management, and interrupt handling.

System Calls

System calls are the means by which a user-level application interacts with the kernel. In a monolithic kernel, system calls are implemented as a set of functions that are called by the application. Each system call serves a specific purpose, such as opening a file, reading from a file, etc. In a microkernel, all these services are provided by components, and thus, only a limited number of system calls are needed.

Syscalls in seL4 and are needed in a microkernel are as follows:

- Call: This system call is used to invoke a function in a component. The component is identified by its TCB. The function to be invoked is identified by its index in the component's function table. The arguments to the function are passed in the IPC buffer.
- ReplyRecv: This system call is used to send a message to a component and wait for a reply. The component is identified by its TCB. The message is passed in the IPC buffer. The reply is also passed in the IPC buffer.
- Send: This system call is used to send a message to a component. The component is identified by its TCB. The message is passed in the IPC buffer.
- NBSend: This system call is used to send a message to a component. The component is identified by its TCB. The message is passed in the IPC buffer. This system call does not block the caller.
- Recv: This system call is used to receive a message from a component. The component is identified by its TCB. The message is passed in the IPC buffer.
- NBRecv: This system call is used to receive a message from a component. The component is identified by its TCB. The message is passed in the IPC buffer. This system call does not block the caller.

- Reply: This system call is used to send a reply to a component. The component is identified by its TCB. The reply is passed in the IPC buffer.
- Yield: This system call is used to yield the CPU to another component.
- A microkernel may also need some thread management system calls, such as create thread, delete thread, etc.
- Syscalls are also implemented to manage virtual memory, such as map, unmap, etc. But a very elementary virtual memory management is implemented in the kernel. The rest of the virtual memory management is implemented in the components, accessed by a IPC call.
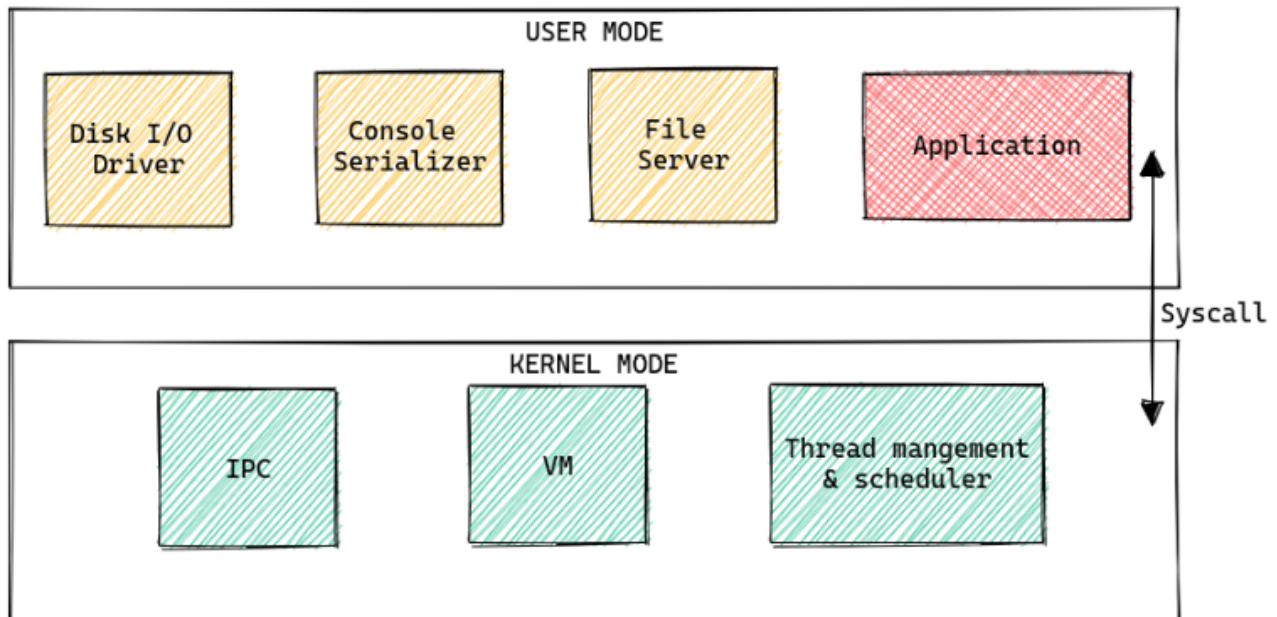
Figure 2: *Example of an microkernel based system*

References

[1] Simon Biggs, Damon Lee, and Gernot Heiser. "The Jury Is In: Monolithic OS Design Is Flawed". In: *Asia-Pacific Workshop on Systems (APSys)*. Korea: ACM SIGOPS, Aug. 2018. URL: `https://ts.data61.csiro.au/publications/csiro_full_text//Biggs_LH_18.pdf`.

[2] Gernot Heiser. *The SeL4 Microkernel: An Introduction*. Tech. rep. 1.2. June 2020. URL: `https://sel4.systems/About/seL4-whitepaper.pdf`.

[3] GEM5. *Gem5 Simulator*. `https://www.gem5.org/`. 2019.

[4] Jochen Liedtke. "On μ-kernel Construction". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, CO, US: ACM, Oct. 1995, pp. 237–250. DOI: `10.1145/224057.224075`.

[5] Gernot Heiser and Kevin Elphinstone. *L4 Microkernels: The Lessons from 20 Years of Research and Deployment*. NICTA and UNSW, Sydney, Australia. 2016. URL: `https://trustworthy.systems/publications/nicta_full_text/8988.pdf`.