Fast, Secure, Adaptable: LionsOS Design, Implementation and Performance

Gernot Heiser, Ivan Velickovic, Peter Chubb
Alwin Joshy, Anuraag Ganesh, Bill Nguyen, Cheng Li, Courtney Darville, Guangtao Zhu,
James Archer, Jingyao Zhou, Krishnan Winter, Lucy Parker, Szymon Duchniewicz, Tianyi Bai
UNSW
Sydney, Australia
{*gernot, peter.chubb, i.velickovic*}*@unsw.edu.au*

Abstract

We present LionsOS, an operating system for security- and safety-critical embedded systems. LionsOS is based on the formally verified seL4 microkernel and designed with verification in mind. It uses a static architecture and features a highly modular design driven by strict separation of concerns and a focus on simplicity. We demonstrate that LionsOS outperforms Linux.

1 Introduction

Safety- and security-critical systems, such as aircraft, autonomous cars, industrial control systems or defence systems, require a highly dependable operating system (OS). The complexity (and code size) of these embedded/cyber-physical systems keeps growing, and it is unavoidable to have highly critical (and, presumably, highly assured) functionality co-exist with less critical (and less trustworthy) functionality. These systems are therefore *mixed criticality systems* (MCS), where the correct operation of a critical component must not depend on the correctness of a less critical component [Barhorst et al., 2009; Burns and Davis, 2017].

Unlike more traditional MCS, which are mostly concerned with (spatial and temporal) *integrity and availability*, the mixed-criticality requirement must also extend to *confidentiality*: Assume an untrusted component of the system is compromised by an attacker, and this component can be used to obtain the system's encryption key used to validate over-the-air software upgrades, then the attacker could replace critical components and thus completely compromise the system.

The core requirement such critical systems impose on the OS is strong temporal and spatial isolation. As such, the seL4 microkernel [seL4 Foundation, 2021a] seems a perfect foundation: seL4 has undergone extensive formal verification, including proofs of confidentiality and integrity enforcement, proof of implementation correctness and proofs that the binary has the same semantics as the verified C code, taking the compiler out of the trusted computing base (TCB) [Klein et al., 2014]. There is furthermore a MCS version of seL4 which provides the temporal isolation properties required by real-time systems [Lyons et al., 2018], verification of the MCS variant is currently in progress [seL4 Foundation, 2024].

These features have resulted in some deployment success, including autonomous military aircraft [Cofer et al., 2018] and, more recently, commercial electric cars [Qu, 2024]. Yet more than a decade after its verification was completed, seL4 is not as widely deployed as might have been expected.

The core reason behind this slow uptake is the low-level nature of seL4, which goes beyond the already low-level nature of other microkernel APIs. For example, seL4 makes all memory management (including for the kernel) a user-level responsibility [Elkaduwe et al., 2008]; while this is a core enabler of reasoning about isolation, the developer experiences it as a foot gun. Together with a lack of support for writing device drivers, this makes seL4 the "assembly language of operating systems". The upshot is that it requires an unusual level of expertise to build functioning and performant systems on seL4 – the kernel is largely unapproachable by industrial developers of critical systems.

In short, to benefit from seL4's provable isolation enforcement, developers of critical systems need an actual *operating system*, providing appropriate abstractions, such as processes, files and network connections, while retaining (and extending) the isolation guarantees provided by seL4. We propose to address this need with LionsOS, designed to meet the following aims:

Fast: LionsOS *performance* should be competitive with that of mainstream systems, such as Linux;

Secure: LionsOS should, by design, not only leverage seL4's verified isolation properties to protect applications from each other, but make an *eventual formal verification* of LionsOS itself tractable;

Adaptable: LionsOS should be deployable on *most critical embedded, IoT and cyberphysical systems*.

These aims are somewhat in conflict: adaptability and verifiability call for a highly modular design. While such a design is natural for a microkernel-based system, the cost of context switches could undermine performance [Chen et al., 2024]. However, in this paper we show that a highly principled (Section 3), modular design (Section 4), combined with an implementation focused on simplicity (Section 5) can result in a system that does meet the performance aim (Section 6).

2 Background and Related Work

2.1 Verification and scalability

The formal verification of seL4 demonstrated that it is possible to prove the implementation correctness of real-world systems of considerable complexity. However, the cost was high: about 12 person years of non-recurring engineering for 8,500 source lines of code (SLOC), and an estimated cost of $350/SLOC [Klein et al., 2009]. While potentially justified for a stable, foundational piece of infrastructure, this cost is too high for most systems, including an OS that will be significantly larger than seL4.

The seL4 project delivered another key insight: Verification effort scales with the square of the specification size [Matichuk et al., 2015]. This implies that there could be a large scalability benefit from keeping things small and simple, i.e. a highly modular design, where each component has a narrow and simple specification, and where module boundaries are enforced by seL4, making it possible to verify modules independently of each other.

While seL4 used labour-intensive interactive theorem proving, recent work increasingly uses automated theorem proving techniques [Cebeci et al., 2024; Chen et al., 2023; Narayanan et al., 2020; Nelson et al., 2017, 2019; Paturel et al., 2023; Sigurbjarnarson et al., 2016; Zaostrovnykh et al., 2017, 2019]. These automated techniques are in essence based on (symbolic) state-space exploration with the help of heuristics to deal with combinatorial explosion. Yet they have severe limitations in the complexity of the code they can tackle, and will generally work best on simple and small modules.

2.2 Modularity in operating systems

The idea of a modular OS with hardware-enforced module boundaries is old, going back to the original microkernel (before the term was coined), Brinch Hansen's Nucleus [Brinch Hansen, 1970]. The approach was popularised by Mach [Rashid et al., 1989] and taken up by other microkernel systems of the time, such as Chorus [Rozier et al., 1988] and QNX [Hildebrand, 1992].

These systems were plagued by poor performance, resulting in moving functionality back into the kernel [Welch, 1991], which did not prevent expensive debacles, such as IBM's ill-fated, Mach-based Workplace OS [Fleisch et al., 1998].

Almost all microkernel-based OSes exhibited course-grained modularity, typically at the level of major subsystems such as file service, networking and process management [Härtig et al., 2005; Herder et al., 2006; Qubes; Rawson, 1997; Whitaker et al., 2002], too large for verification. Nevertheless, even the most recent work argues that the cost of crossing module boundaries is too high, resulting in co-locating services into even more course-grained isolation domains [Chen et al., 2024].

The Flux OSKit [Ford et al., 1997] was an early design featuring a more fine-granular design. Performance comparison to Linux and FreeBSD showed a 13% degradation in networking throughput and a 45% increase in latency. SawMill was an ambitious project aiming to break up Linux into components isolated by a microkernel, file-system benchmarks showed a throughput degradation of about 18%. No CPU load values are reported for any of these systems, but the degradation in achieved throughput indicates a significant increase in per-packet processing cost.

Genode (formerly called Bastei) [Feske, 2015; Feske and Helmuth, 2007] features a modular design aiming for assurance and explicitly prioritising this over performance (we could not find any published performance data). Its implementation in C++ will prevent a complete formal verification for the foreseeable future.

An alternative is modularisation enforced by programming languages (as opposed to address-space isolation mediated by a microkernel), as pioneered by SPIN [Bershad et al., 1995], and later adopted by Singularity [Fähndrich et al., 2006] and RedLeaf [Narayanan et al., 2020]. These systems generally exhibit lower performance than mainstream OSes. Furthermore, as their security relies on type-safety enforcement by the programming language, they require the whole OS to be implemented in that language. Inevitably this requires unsafe escapes for dealing with hardware. More importantly, this rules out re-using code from mainstream OSes.

Writing all device drivers from scratch is generally infeasible. We therefore ignore language-based isolation approaches and instead focus on modularity enforced by address-space isolation.

3 LionsOS Design Principles

Given this experience, what makes us think we can meet all our aims from Section 1?

We observe that a commonality of these earlier systems is a significant complexity in design and implementation. We posit that the key to meeting our aims is a strict application of the time-honoured *KISS Principle* [Wikipedia, 2001]. Following this high-level principle, we aim for a highly modular design which incorporates the following secondary principles:

Strict separation of concerns: Each module has one and only one purpose (as far as feasible). Furthermore, a particular concern (e.g. the traffic-shaping policy of a network subsystem) should be fully contained in a single module.

Least privilege: Each module only has the access rights it needs, not more. While not strictly a consequence of KISS, this time-honoured security principle of Saltzer and Schroeder [1975] will simplify reasoning about security and safety of the system.

Design for verification: Given the scalability and complexity limits of verification (Section 2.1), this principle calls for keeping module interfaces narrow and module implementations simple.

Use-case specific policies: This is arguably the most controversial principle, as it calls for tailoring each (resource) policy to the system's particular use case, in order to simplify the policy implementation.

It seems clear that adhering to these principles, which we summarise as "radical simplicity", will give us the best chance to produce a highly dependable system and maximize the chances of formally proving its correctness and security.

Simplicity is aided by restricting our target domain to embedded systems. While aiming for generality within that domain – which specifically includes cyber-physical systems such as autonomous aircraft and cars, some of which are quite complex and demanding – we do not (yet) aim to support more general-purpose systems, such as cloud servers, smartphones and certainly not laptops.

The common thread of the embedded domain is that it can be served with a *static system architecture*, i.e. a set of components that is known at configuration time. Note that this does not mean that the system is fully static, it can still support late loading of components, dynamic component updates, and place-holders for components that can be instantiated with programs not known at build time. Dependable embedded systems generally cannot over-commit resources, which is what makes the static architecture work. We are yet to see a realistic use case in the embedded space that cannot be addressed with a static architecture.

The inherent constraints of embedded systems are also in line with the principle of use-case specific policies. A computer system generally has two classes of policies: security and resource policies. In the embedded space, the security policy is generally defined by the use case, and will only change in the context of a significant reconfiguration of the system (accompanied with a major software upgrade).

But the embedded system's defined set of resources implies that, at least for the critical systems we are targeting, the designer also has (or should have) a clear idea of how they should be managed. The more tailored the policy is to the use case, the simpler its implementation, and the easier it is to assure (formally or informally) that it matches requirements.

This principle of use-case specific policies is arguably the clearest departure from conventional approaches. OSes tend to be designed to adapt to changing application scenarios with no or minimal code changes. This naturally leads to a (whether conscious or not) desire to provide *universal policies*. Of course, no policy is truly universal, and sooner or later it will encounter a use case where the existing policy behaves pathologically, resulting in attempts to generalise it. This approach is a massive driver of complexity: For example, the Linux scheduler contains five scheduling classes, each of which has one or more per-thread tuning parameters; the scheduler has grown from around 11,kSLOC in 2011 (version 3.0) to over 30 kSLOC today (version 6.12). Furthermore, the approach frequently leads to optimising particular "hot" use cases at the expense of overall performance [Ren et al., 2019].

Use-case specific policies represent the opposite approach: Each policy is highly specialised for the use case, and the system achieves use-case diversity not by generalising policies, but by re-writing them as needed. Of course, this can only work if the policies are simple enough to implement.

Our underlying argument is that by taking a radical approach to simplicity and use-case specificity, policies *do* become simple. Moreover, for most resources there is a small to moderate set of policies that can be pre-supplied, letting the system designer chose from an existing set (or trivially adapting an existing one to the use case).

An illustrative example is network traffic shaping: If multiple clients of a network interface overload the interface, there really are a small number of obvious policies to choose from: Clients may be given a priority, their bandwidth may be limited, or they may be served round-robin. And in an embedded system, it is usually obvious which one is appropriate, and this is unlikely to change for the lifetime of the system.

Taken together, our principles lead to an OS becoming something akin to a Lego® set: The OS is built from different kinds of components (brick shapes). Each component comes in multiple, functionally compatible versions (brick colours), and the choice of version (colour) can be made largely independent of the rest of the system.

An obvious concern is how the fine-granular modular-

ity affects our performance aim, as this necessarily leads to high context-switching rates. Given there is a cost to every context switch (depending on the architecture, 400–600 cycles for seL4 [seL4 Foundation, 2021b]), this has the potential to make the system slow [Chen et al., 2024]. We will examine the performance impact in Section 6.

4 LionsOS Design

Like most OSes, LionsOS consists mostly of I/O subsystems (device drivers, protocol stacks, file systems) and resource management. The latter part is particularly small in LionsOS because of its static architecture, which reduces resource management to some simple policy modules that are not only use-case specific (as per our principle) but also local in their nature (e.g. shaping traffic of a network interface).

Some global resource management is required but not yet fully designed (nor implemented), such as core management (off- and on-lining processor cores based on overall CPU load). We will discuss this in Section 5.3.

4.1 Devices

We demonstrate how our principle of strict separation of concerns applies to I/O, using networking, specifically Ethernet, as a case study. We briefly discuss other device classes in Section 4.1.5.

4.1.1 Device drivers

Applying separation of concerns, we reduce the purpose of a device driver to *translate between a hardware-specific device interface and a hardware-independent device class interface*. Hence an Ethernet driver abstracts the specific NIC as a generic Ethernet device.

Unsurprisingly, the Ethernet device-class interface (i.e. the driver's *OS interface*) looks similar to that of an actual Ethernet NIC, with some differences that help simplify its use. NICs typically use ring buffers in DMA memory to pass references to DMA buffers from and to the driver; each ring buffer entry contains a pointer to a buffer in the DMA region, together with some meta-data (indicating whether a buffer contains valid data). A NIC usually references two such ring buffers, one for transmit (Tx) and one for receive (Rx) data.

To simplify the OS interface, we separate queues for buffers containing valid data from those that do not. This means that the SW side of the driver has four queues:

transmit available (TxA): references buffers provided by the OS to be sent to the network;

transmit free (TxF): references buffers returned by the NIC to the OS for re-use;

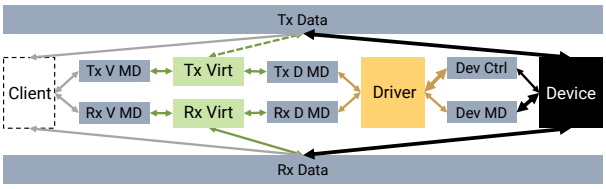

Figure 1: Memory regions for Ethernet: device control (Dev Ctl), device metadata (Dev MD), transmit and receive data (Tx Data, Rx Data), driver metadata (Tx D MD, Rx D MD) and virtualiser metadata (Tx V MD, Rx V MD). Arrows indicate access to regions by components, thick black arrows indicate DMA by the device, the thick coloured arrow indicates uncached access by the driver. The TxVirt only maps the Tx Data region if needed for cache management (shown as dashed). We only show a single client of the virtualiser.

receive available (RxA): references buffers filled with data by the NIC to be consumed by the OS;

receive free (RxF): references buffers provided by the OS to the NIC to receive data from the network.

These queues are allocated in *driver metadata regions*, for Ethernet there is one for Tx data (containing the TxA and TxF queues) and a separate one for Rx data (RxA and RxF queues). These are "normal" memory invisible to the device (i.e. not accessed by DMA).

Note that the driver only handles *pointers* to DMA buffers, it has no need to access the actual data. We make this explicit by separating the *data region*, which contains the buffers to be filled/emptied by the device, from the *device metadata region*, which contains the ring buffers pointing to the data buffers. In addition there is the *device control region*, which maps the device registers for memory-mapped I/O. Data and device metadata regions are memory regions accessed by the device via DMA.

As the driver has no need to access the I/O data, it *does not have the data region mapped* into its address space, in line with the principle of least privilege. The device control region is mapped to the driver *uncached*.

The right half of Figure 1 shows the memory regions relevant to the driver. For the Ethernet device class, we treat the Tx and Rx paths separately, hence the driver's OS interface has two metadata regions, one for transmit and one for receive.

4.1.2 Virtualisers

While the driver abstracts the device hardware, it does not deal with address translation, nor with sharing the device between multiple clients. This is the purpose of a separate *virtualiser* (Virt) component, which:

- shares a physical device between multiple clients, presenting each with a virtual device;

- translates references to DMA buffers from client addresses to device addresses (physical addresses or IOMMU-translated I/O-space addresses);
- performs cache management (flushing/invalidating) where needed (the x86 architecture keeps caches coherent with DMA and thus does not require this functionality).

For the Ethernet device class, we have independent virtualisers for the Tx and Rx paths (TxVirt and RxVirt).

The Virt replicates its driver interface at the client side, meaning its client-side interfaces look exactly like its device interface. Specifically it has a metadata region (Tx V MD, Rx V MD) that structurally replicates the respective device metadata region. The key difference is that while the device MD regions use I/O addresses for referring to data buffers, the Virt MD regions refers to data buffers by *offsets* from the beginning of the respective data region, thus making the Virt's address-translation task independent of client virtual address-space layout.

The RxVirt needs to inspect the headers of incoming packets and thus requires the data region to be mapped (R/O). The TxVirt does not need to access the data region directly, but may need to have it mapped in its address space to perform cache management. For example, the Arm architecture performs cache operations on virtual address ranges, so the TxVirt needs the data region to be mapped into its address space. Arm does not have cache-coherent DMA, so memory that will be transferred to a device using DMA has to be cache-cleaned before DMA occurs. Likewise, the RxVirt on Arm needs to invalidate caches after DMA into its buffers. These mappings are indicated in the left half of Figure 1.

The TxVirt must implement a *traffic shaping* policy if its clients generate load that exceeds the NIC's Tx capacity. In line with our principle of use-case specific policy, the Tx policy is as simple as the specific use-case allows. It is typically one of round-robin, priority-based or bandwidth limiting.

The RxVirt may only require a simple policy: what to do when data arrives for a client whose RxA queue is full, possible choices are to block or (more likely) discard the packet and return the buffer to the driver's RxF queue. We generally avoid this this need by ensuring that all client queues are large enough to hold all available Rx buffers, starving the device of buffers if the clients fail to process input fast enough – this leads to the NIC dropping packets under overload without wasting CPU cycles, and leaves the RxVirt policy-free.

4.1.3 Data regions and copiers

The Tx data region is seen as a single region by the driver. However, each client has its own sub-region, which is mapped into the client address space (and the Virt's where required).

While each client data region is contiguous in physical memory, there is no need for contiguity of the overall data region. Obviously, the whole region must be mapped to the device by the IOMMU.

The same approach does not work for Rx data, as the device will deposit input in any free buffer, and only the Virt determines the target address space. There are three possible approaches to making Rx data available to the correct client:

1. have only a single, global Rx data region, and the Virt maps each buffer to the client when inserting it into the client's RxA queue, and unmapping it when retrieving a buffer from the client's RxF queue. This needs additional privilege in the Virt, but the Virt must be trusted anyway;
2. have only a single, global Rx data region, which is mapped R/O in all client address spaces. This implies that clients can read other client's input data;
3. have an explicit *copier* (Copy) component between the RxVirt and each client, which copies the data from the global data region into a per-client data region.

The actual choice comes down to performance (incl. a trade-off between the cost of copying and the cost of mapping operations) and the system's security policy. The advantage of the model is that the decision can be made without affecting the implementation/operation of either the driver or the clients. For example, option (2) is suitable if there is no concern about one client seeing another's input data (e.g. when all network traffic is encrypted). The Copy component can be inserted transparently: the difference between case (2) and (3) above does not affect the implementation of either the Virt or the client/copier.

4.1.4 Broadcast

We offer two schemes for handling address resolution request (ARP) packets coming from the network.

The first approach uses a separate ARP client, whose only job is to respond to those requests. This requires that incoming traffic is routed to clients based on MAC address, and that the ARP client has (R/O) access to the MAC-address allocation table.

The alternative approach handles broadcast packets in the RxVirt by enqueuing a copy of the ARP packet in each client's queue, and reference counting the driver's buffer containing the packet. We return the buffer to the driver's RxF queue once each client has enqueued its copy in its RxF queue (and thus the reference count has dropped to zero). Each client is then responsible for handling any broadcast traffic it receives.

4.1.5 Other device classes

Some other device classes look similar to Ethernet at a high level, and result in a similar design. This includes most serial devices (serial ports, SPI, I2C) with differences in the details for the protocol. Some have no separation between data and metadata (the queues directly contain the data).

Others, especially storage devices, do not have the clear separation of Tx and Rx traffic of Ethernet, and instead, the storage device only reacts to explicit read or write requests. This results in a slightly different design:

- there is a single driver metadata region;
- there are only two queues, the *request* (Rq) and the *response* (Rs) queue;
- there is a single Virt, which presents an Rq and Rs queue to each client in a per-client metadata region
- there is one data region per client.

In addition to read and write requests, the Rq may also contain *barrier* requests, across which the device is not allowed to reorder other requests. Other protocol details support batching of requests. The storage Virt statically partitions the devices between its clients.

The driver exports an *information page* of device properties, which is appropriately virtualised by the Virt.

4.2 OS services

For best performance, LionsOS presents a *native* API that is asynchronous and modelled largely on the device interfaces. For developer convenience and to ease porting of legacy applications, LionsOS also provides a blocking, POSIX-like API that is layered over the native one.

As network traffic is explicitly (de)multiplexed by the virtualisers, there is no need for a global IP stack, it becomes a library linked directly into the client. This takes the complex (and probably buggy) protocol stack out of the system's TCB.

The same approach can be used for storage, by providing a per-client file-system library that directly operates on the virtual storage device provided by the Virt (with an optional copier in between). Alternatively, a single (trusted) file system can be used for all clients, which then is the sole client of the storage Virt.

Sharing across per-client file systems can be enabled by an explicit multiplexer component that connects to multiple clients.

A unified view of multiple storage devices can be provided to a client through a virtual file system that interfaces to multiple per-device file systems. However, multiple storage devices are rare in the embedded space.

Figure 2 shows these options. The main take-away is that separation of concerns means that such choices come down to selecting the appropriate components from the "Lego" set and combining them as needed.

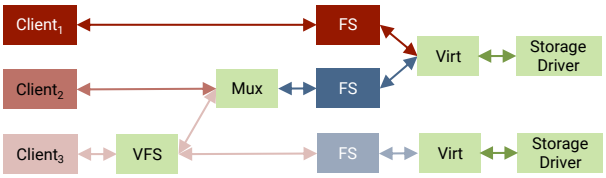

Figure 2: File system configuration options.

5 LionsOS Implementation

5.1 The starting point: seL4 Microkit

We base the design of LionsOS on the seL4 Microkit [seL4 Foundation, 2023] (formerly "Core Platform"). The Microkit simplifies seL4 usage by imposing a static system architecture and an event-driven programming model. It presents an abstraction of the seL4 API that is partially verified using SMT solvers [Paturel et al., 2023].

The Microkit provides a process abstraction called *protection domain* (PD). PDs are single-threaded and combine the seL4 abstractions of virtual address space, capability space, thread and scheduling context. Multi-threaded processes can be implemented through multiple PDs that share an address space. While useful for applications, we do not use this for LionsOS itself – all LionsOS components are strictly sequential.

PDs communicate via shared memory and semaphores (seL4 notifications). Server-type PDs can be invoked synchronously via *protected procedure calls* (PPCs), which map onto seL4 synchronous IPC – such a server executes on the caller's core.

PDs are structured as event handlers, signalling their semaphore will execute the `notified` function, identifying the sender PD. A server has another handler function, `protected`, to receive PPCs. Each PD also has an `init` handler for initialisation.

The system architecture of PDs and their communication channels (semaphores and shared memory regions) is defined in a *system description file* (SDF). It specifies the ELF files to be loaded into each PD and a PD's metadata, including scheduling parameters, access rights to memory regions and caching attributes, and access to interrupts (which appear as semaphores). A PD can monitor a *virtual machine*, in which case it acts as a private virtual-machine monitor (VMM) which handles virtualisation events from the VM.

Microkit tooling generates from the SDF the seL4 system calls that set up the PDs, channels and memory regions and invokes each PD's `init` function. The tooling

hides the complexities of seL4's capability system from the developer.

5.2 Queues and state

The design using explicit virtualisers (for separation of concerns) enables another important simplification: All queues are single producer, single consumer (SPSC), lending themselves to simple, lock-free implementations. Specifically, the TxF and RxA queues hold data that is provided by the driver (original producer) and consumed by the client (ultimate consumer), packets flow from right to left in Figure 1; for the TxA and RxF queues the flow is in the opposite direction.

The queues are also inherently bounded, leading to a simple, array-based implementation, where references to particular queue entries are simply array indices. We also require all queues to be a power of two in size, further simplifying implementation and sanitation.

An important property of this design is that all policy-independent state is held in shared memory. This makes it easy to restart a failed component without affecting the rest of the system (other than by a transient latency glitch). This even enables switching policies on the fly, by reloading the code of a component. We demonstrate this in Section 6.3.1.

5.3 Location transparency

In standard producer-consumer fashion, the lock-free SPSC queues are synchronised by semaphores (signalling a Microkit channel). A producer component signals the consumer if new buffers have been enqueued, and the consumer has indicated that it requires signalling by setting a flag that resides in shared memory. A consumer chooses whether or not to set this flag based on if there is meaningful work that it can do with new buffers. In most of our sub-systems it is more common for a producer to be signalling a consumer, but a consumer can also signal a producer if it has done meaningful work, and if the producer has indicated that it needs signalling by setting its shared flag. This can be useful when a producer wishes to enqueue more data in a queue which is currently full.

This approach is completely *location transparent*: A particular component is not aware whether the component with which it shares a queue is running on the same or a different core. This location transparency of components makes up for the strictly sequential nature of Microkit components: Instead of requiring error-prone, multi-threaded implementation of components to make use of multicore hardware, LionsOS utilises multicore processors by distributing components across cores.

The result is that *concurrency is tamed*: almost all code is freed from concurrency control. The only requirements are correct use of semaphores and flags, and the correct implementation of the enqueue/dequeue library functions (which are straightforward due to the SPSC nature of the queues).

Location transparency will also simplify core management: If a core needs to be off-lined, components running on it can be transparently migrated to other cores, without affecting the system's operation (other than some temporary latency increases). However, we have not yet implemented this.

5.4 Legacy driver reuse

The LionsOS design vastly simplifies drivers compared to other OSes (see Section 6.2), and implementing drivers from scratch will result in the best performance.

However, it is unrealistic to expect adopters to write all drivers from scratch, especially since in practice few devices are performance critical enough to justify such an effort. It is also frequently impractical, as many devices are poorly (or un-)documented. For such cases, LionsOS allows reusing a driver from Linux by encapsulating it in a virtual machine (VM). Unlike the Dom0 driver VM of Xen [Barham et al., 2003] and used in other microkernel systems [Chen et al., 2024], we follow the approach of LeVasseur et al. [2004] and wrap each driver in its own VM.

Figure 3 shows the architecture. The driver VM runs the legacy driver as part of a (minimally configured) Linux guest. The guest runs a single, statically-linked usermode program, the *UIO driver* (which replaces `init`). The program uses normal Linux system calls to interact with the device, and the Linux user I/O (UIO) framework to interact with the LionsOS driver queues.

Specifically we use UIO to map guest physical memory (to access the queues) and receive virtual interrupts. seL4's virtual machine architecture re-directs virtualisation exceptions to a per-VM virtual-machine monitor. We use this to inject semaphore signals from the Virt as IRQs into the VM, to be received by the UIO driver.

We supply the driver VM's complete userspace as a CPIO archive loaded at boot time from a RAM disk.

5.5 Implementation status

5.5.1 Device drivers

Most of our development happens on the HardKernel Odroid-C4 (Amlogic S905X3 SoC) and the Avnet MaaXBoard (i.MX8MQ SoC) platforms, so this is where we have the largest set of native drivers. Specifically, we have native drivers for the following device classes:

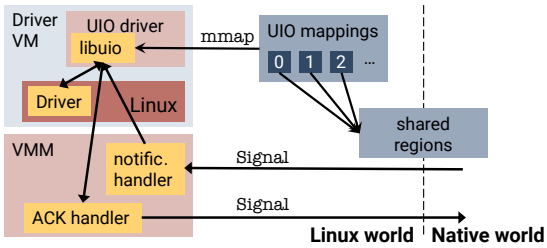

Figure 3: Driver-VM architecture

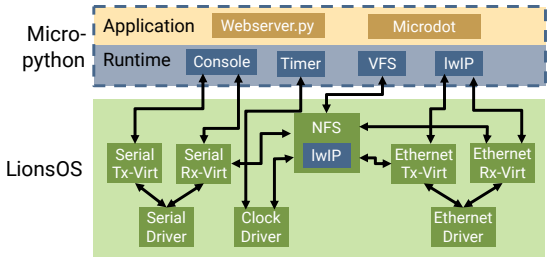

Figure 4: Architecture of the LionsOS-based web server.

- Serial for all supported platforms.
- PinMux and clock for MaaXBoard and Odroid-C4.
- Ethernet for Odroid-C4, MaaXBoard and the i.MX8 series FEC.
- Block: SDHC drivers for the MaaXBoard and Odroid-C4 – the latter written in Rust.
- VirtIO drivers (for running on top of QEMU) for serial, block, network and graphics (2D).
- an I2C host driver for the Odroid-C4.
- I2C drivers (using the I2C host driver) for a PN532 NFC card reader and a DS3231 real-time-clock.

These are written in C unless otherwise stated, but the system does not prescribe an implementation language, as demonstrated by the Odroid-C4 SDHC driver being written in Rust.

We have Linux-based driver VMs for the following device classes:

- GPU via exported framebuffer for Odroid-C4.
- Ethernet for Odroid-C4.
- Block (SDHC) for the Odroid-C4.
- Sound using the ALSA framework for Odroid-C4.

5.5.2 Services

At present we have full networking functionality, using lwIP [Dunkels, 2001] as a client library. Both the native (synchronous) as well as the blocking API are supported, the latter is layered over the former using a coroutine library. We also have an NFS client (using an open-source NFS library) which uses the blocking API. For address-space separation (Section 4.1.3) we use an optional Copy component in the Rx path.

We have an asynchronous filesystem API that is used to talk to a component running NFS or to a VM. The Linux guest in the VM uses the standard VirtIO block interface (converted to our block interface by the VMM), and can make any filesystem supported by Linux available to other components using the filesystem API.

For security, the VM-based filesystem need not be shared between components. Block device sharing is then by partition and enforced by the block virtualiser. Particular use cases can allow more sharing if desired, by using a component between the filesystem and its clients to check allowable access. The VM-based Linux filesystem is not amenable to any kind of formal verification.

The system is complete enough to run several complete systems in daily use. One of them is a reference design for a point-of-sales terminal. It uses a driver VM for re-using the Linux GPU driver, and either a native Ethernet driver or another driver VM for re-using a Linux driver (to demonstrate multiple driver VMs). Another deployed system is a web server that hosts the sel4.systems web site.

The web server has the business logic implemented in Python, supported by a port of MicroPython [Developers] to LionsOS, Figure 4 shows the architecture.

5.5.3 Resource management

We have not yet implemented dynamic resource management, such as core off-/on-lining (cf. Section 5.3), but plan to do this within a year.

6 Evaluation

We evaluate multiple aspects of LionsOS, covering development and debugging effort, legacy driver re-use, and performance.

6.1 Platforms

LionsOS supports the Arm AArch64, Intel x86_64 and RISC-V RV64 architectures. Development happens primarily on Arm platforms and is then ported to the other architectures. We evaluate on Arm and x86.

The Arm platform is an Avnet MaaXBoard with an NXP i.MX8MQ SoC, having four Cortex A53 cores running at a maximum of 1.5 GHz; we run our measurements at a clock rate of 1 GHz. It has 2 GiB of RAM, an on-chip 1 Gb/s NIC, and an on-chip SDHC controller. We perform Linux measurements on this board with a small Buildroot system using kernel version 6.1.0.

The x86 platform is an Intel Xeon® W-1250 six-core CPU running at 3.3 GHz, with an Intel IXGBE X550 10 Gb/s copper NIC. We disable hyperthreading and turbo-boost. For Linux measurements we use a Debian

Bullseye userspace and use the "performance" CPU frequency controller to keep the CPU frequency at its maximum value.

6.2 Complexity and development effort

6.2.1 Code size

Our subjective experience is that the LionsOS model dramatically simplifies development of core OS components. A striking example is the i.MX8 network driver, which was the first device driver written to the LionsOS driver model of Section 4.1.1. It was implemented by a second-year undergraduate student, less than 18 months after she wrote her first program. Table 1 compares its size with the Linux driver for the same device. It also compares drivers we use on our x86 platform. We use `sloccount` [Wheeler, 2001] for all code-size measurements.

The size gives an indication of the complexity of the task. The student found she was spending very little time in debugging the driver logic, unlike normal driver development.

Our experience with other LionsOS components is similar, components are small and simple. Table 2 gives a breakdown of the point-of-sales terminal we developed as a demonstrator (see Section 5.5.2). LionsOS, as configured for this application, consists of about 3.1 kSLOC of trusted code, plus 66 kSLOC of untrusted library code that cannot break isolation. All the trusted code is written from scratch.

6.2.2 Signalling protocols

While the LionsOS design clearly reduces overall complexity, it does shift some complexity into the inter-component synchronisation protocols. A pessimistic implementation of those protocols is trivial: the producer notifies the consumer whenever it inserted something into a shared queue, and the consumer signals whenever it took something out of a queue. This clearly leads to over-signalling and should be avoided.

A potential refinement is to signal only if there is a substantial change to a queue: The producer signals only if the queue becomes non-empty, the consumer only if the queue becomes non-full. Implementing such a protocol correctly however can prove to be difficult, as checking the fullness status of a queue modified by two components in shared memory can introduce race conditions. The components can obtain disagreeing measurements, causing both unnecessary signals or failures to signal at critical times. As well as this, the design can still result in over-signalling as a component's ability to make progress is usually not dependent on only one queue.

As an example, consider the Ethernet driver. It removes active buffers from the device Rx ring and immediately inserts them into the Virt's RxA queue. It also removes buffers from the Virt's RxF queue and inserts them into the hardware ring. This means that the Rx code of the driver can only make progress if one of two conditions are satisfied:

1. The hardware Rx ring is non-empty and has active buffers **and** the RxVirt's RxA queue is non-full
2. The Virt's RxF queue is non-empty and the hardware Rx ring is non-full.

This means that if the Virt signals the driver whenever the RxA queue becomes non-full or the RxF queue becomes non-empty, it may wake the driver only for the latter to find that the hardware Rx ring is not in the necessary state to make progress and go back to sleep.

We address this by letting the driver indicate with a flag for each queue whether it wants to be woken. This allows the driver to incorporate its knowledge of the hardware ring state into the Virt's decision of whether to signal. However, the resulting increase of complexity of the intertwined protocols is a source of subtle concurrency bugs that can lead to deadlocks.

Fortunately we find that model-checking is a useful tool to alleviate these protocol bugs, and we are able to create models of each of our components using SPIN [Holzmann, 1997], and use these models to prove deadlock freedom of our protocols. This allows us to optimise each component's decision to signal more aggressively,

Platform	Speed	Linux	LionsOS	Ratio
i.MX8	1 Gb/s	4,775	569	8.4
x86	10 Gb/s	3,019	668	4.5

Table 1: Code sizes in lines-of-code of LionsOS drivers compared to Linux.

Component	**LoC**	**Library**	**LoC**
Serial Driver	249	Microkit	303
Serial TxVirt	175	Serial queue	219
Serial RxVirt	126	I2C queue	101
I2C Driver	514	Eth queue	140
I2C Virt	154	Filesys queue & protocol	268
Timer Driver	136		
Eth Driver	397	Coroutines	848
Eth TxVirt	122	lwIP	16,280
Eth RxVirt	160	NFS	45,707
Eth Copier	79	VMM	3,098
Total	2,112	1,031 + 65,933	

Table 2: Code-size breakdown of the point-of-sale terminal demonstrator. Components in red font are not part of the LionsOS TCB. The lwIP and NFS libraries ported from other systems, the rest is written from scratch.

as we can be assured that no critical signals are missed.

6.2.3 Driver VMs

Contrary to claims made by Chen et al. [2024] (which are based on work by LeVasseur et al. [2004] that preceded hardware virtualisation support), we find that reusing a Linux driver in a virtual machine is an easy way to gain access to a device. The main challenges are platform initialisation, and avoiding excessive resource consumption at runtime.

Effort During the development process, the VM image can include standard Linux development and debugging tools, supporting rapid development of the UIO driver, typically within a few days. Once built, that UIO driver will work for *any* device of the class it has been built for.

The resulting engineering effort per device-class is small (a few weeks). The main disadvantage (apart from the relatively large size of an entire VM for a single driver) is increased latency in accessing the device.

Limiting resource usage RAM usage can be limited by carefully configuring kernel and userspace to use only what is needed (see Table 3). The size differences between the unoptimised GPU and the optimised audio and block device VMs show the order-of-magnitude gains possible with a bit of effort. The resulting memory footprints are small given today's memory sizes. However, complex SoCs may contain hundreds of devices, so when using a separate VM for each driver, the space cost will add up. We can amortise this cost over multiple devices by supporting multiple drivers from a single VM, at the expense of reduced isolation between drivers – likely a sensible approach for the myriad of simple devices on a complex chip.

Platform initialisation Most devices need various platform registers to be set appropriately, to set up clocks for the device, and connections to the outside world (pins). We use native PinMux and clock drivers to set up the platform to enable all devices in the device tree. The driver VM traps accesses to the PinMux and clock registers, discards writes, and passes the actual values in the registers to the native drivers, which provide an interface for querying register values. For debugging, the native drivers support printing the values passed to them.

Driver	Kernel	RAM disk	Runtime
GPU	33 MiB	6.4 MiB	128 MiB
Audio	3 MiB	2.4 MiB	18 MiB
Block	3 MiB	48 KiB	12 MiB

Table 3: Sizes of driver VMs (GPU is **not** optimised).

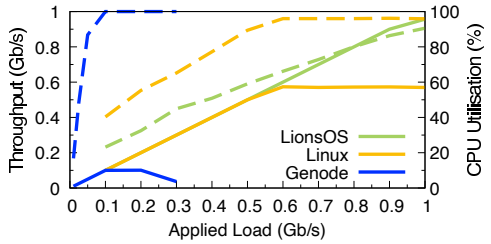

Figure 5: LionsOS vs Linux performance on i.MX8MQ UDP echo benchmark, single core. Solid lines are throughput, dashed lines CPU utilisation

Some devices require changing PinMux or clock settings at run time. For example Odroid C4 uses a pin either as a clock output, or as a GPIO, while upgrading an SDHC card from its initial low-speed to a high-speed state. Supporting such dynamic setting is future work.

6.3 Performance

6.3.1 Networking

To measure networking performance, we configure the evaluation system (LionsOS or Linux) with an *echo* client, which receives packets from the network and immediately sends them back. The evaluation system also measures CPU load. For LionsOS we use the lwIP protocol stack as a library, linked against the client, and an RxCopy.

We use an external load-generator machine, connected to the evaluation platform via a switch. The load generator runs `ipbench` [Wienand and Macpherson, 2004] to apply a varying load of UDP traffic and measures the achieved throughput, i.e. the bandwidth received back from the evaluation system. `ipbench` also measures round-trip time (RTT).

We compare against Linux Debian 6.6.15-2 (2024-02-04) running the standard in-kernel IP stack.

Figure 5 shows the result on the Arm platform, where everything is pinned to a single core. The LionsOS system has no problem keeping up with the load, up to the capacity of the 1 Gb/s NIC, while Linux plateaus at just under 600 Mb/s throughput. Looking at the CPU load we see the reason: Linux maxes out the core at an applied load of 600 Mb/s, while LionsOS achieves the full Gb/s throughput with about 10% CPU capacity left spare. In other words, LionsOS uses just over half the CPU of Linux for handling the same load.

We also run this benchmark on Genode 24.11 [Feske, 2015], using the "hw" base platform, which runs bare metal without a microkernel underneath. We configure Genode with three components: the echo server (using lwIP), the uplink component (similar to our Virts), and the NIC driver, which is transplanted from Linux 6.6.47.

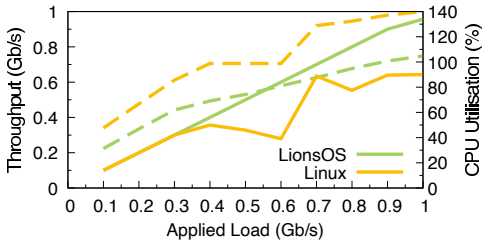

Figure 6: LionsOS vs Linux performance on i.MX8MQ UDP echo benchmark, SMP.

We take all directly from the Genode source tree, compose into a single system with a custom run script based on the `mnt_pocket_stmmac_nic` example.

The Genode throughput is the blue line in Figure 5. It reaches a maximum of about 100 Mb/s throughput, where it maxes out the CPU, and then collapses at 300 Mb/s applied load. Given this disappointing result, we also run netperf benchmarks on the netperf_lwip example provided by Genode. The TCP_STREAM and TCP_MAERTS benchmarks, which measure unidirectional TCP performance, achieve around 250Mb/s and 230Mb/s respectively, still far below throughputs achieved by either Linux or LionsOS.

Figure 6 shows multicore results on Arm. The story is similar to unicore: Linux fails to handle the full load (but only uses 1.4 cores at maximum load) while LionsOS handles the full load with just over 1.0 cores. CPU load is higher than on unicore as cross-core signalling is more expensive than intra-core.

We perform the comparison to Linux on the x86 platform as well, Figure 7 shows unicore results. LionsOS handles the traffic up to an applied load of about 6 Gb/s despite almost maxing out the CPU at a load of 4 Gb/s, after which it can no longer keep up. However, throughput still increases, maxing out at about 7 Gb/s. The continued increase in throughput results from natural batching: the driver finds an increasing number of packets to process per IRQ.

In contrast, Linux, while similarly maxing out the

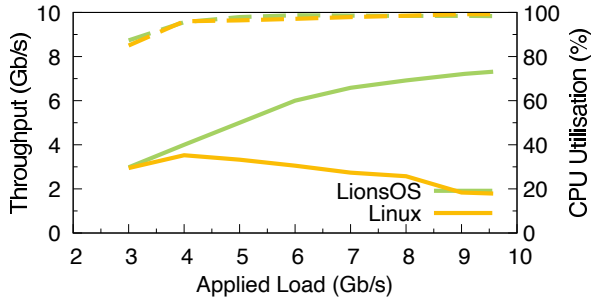

Figure 7: LionsOS vs Linux performance on x86_64 UDP echo benchmark, single core.

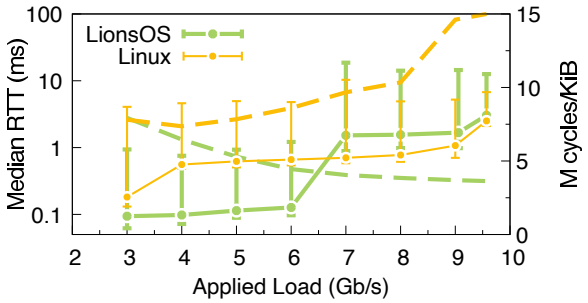

Figure 8: Single core median round-trip times on x86_64 for LionsOS and Linux, bars show observed range. Dashed lines represent the CPU cost per unit of data.

CPU at around 4 Gb/s, shows a performance collapse at increasing load, dropping from a maximum throughput of 3.5 Gb/a to less than 2 Gb/s at maximum applied load.

Figure 8 compares round-trip times for the two systems. LionsOS RTT is consistently low (below 200 μs) until it increases sharply when the load exceeds 6 Gb/s. In contrast, Linux' RTT is consistently much higher, around 1000 μs across the range. Looking at the CPU cycles per KiB processed, LionsOS and Linux start out the same but then diverge, as LionsOS per-packet processing cost drops (due to batching) with increasing load, while Linux' cost increases.

As can be see from Figure 9 LionsOS can achieve full network bandwidth using two cores, whereas Linux achieves only 4Gb/s (but avoids a performance collapse).

IPC cost We now perform an experiment to investigate the impact of the cost of microkernel IPC operations. Mi et al. [2019] compare IPC costs between seL4, Fiasco.OC and Google's Zircon microkernel on an x86 platform. They find intra-core fastpath round-trip IPC latencies for the three kernels to be 986, 2717 and 8157 cycles respectively. So we simulate those kernels by adding $(2717-986)/2 = 865$ ("Fiasco") and $(8157-986)/2 = 3585$ ("Zircon") respectively to each seL4 system call (by executing a tight loop while moni-

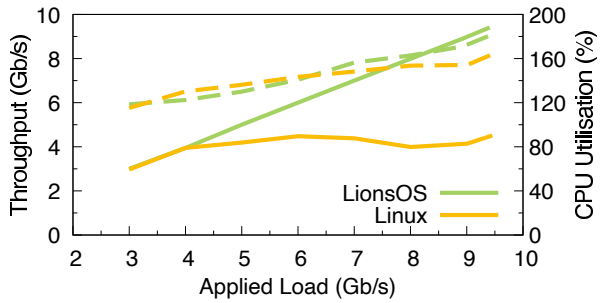

Figure 9: LionsOS vs Linux performance on x86_64 UDP echo benchmark, SMP.

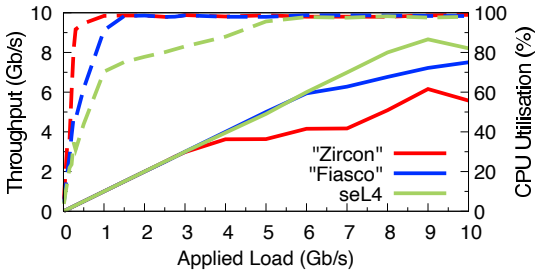

Figure 10: Single core throughput and CPU utilisation on x86_64 for seL4-based compared with simulated Fiasco- and Zircon-basedLionsOS.

toring the cycle counter). We do not claim that this is in any way precise, but it should give a rough estimate of how the same setup will perform on those kernels.

Figure 10 shows the results. We see that the seL4-based LionsOS has lower CPU usage and achieves higher throughput than the Fiasco simulation. Simulated Zircon only achieves about 2/3 of the seL4 throughput.

Virtualiser policy Figure 11 shows an experiment with three clients, to which the TxVirt assigns different policies (high, medium, low). We vary the load on the medium-priority client and impose no load on the others, other than pinging. The graph shows ping latencies of the three clients as a function of load on the medium-priority client. The priorities are reflected in the ping times being fastest on the high-priority, and slowest on the low-priority client.

Dynamic policy swap We investigate swapping policies at run time. To demonstrate this we configure a system with a TxVirt that monitors bandwidth, plus a *Swapper* that can reload the TxVirt with a different executable, a bandwidth limiter. When a preset throughput threshold is exceeded, the Virt signals the Swapper, which the loads the new program into the Virt. (This setup uses experimental Microkit features.)

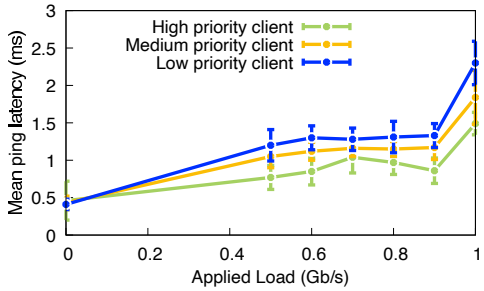

Figure 11: Ping latencies on system with three prioritised clients as a function of load on the medium-priority client on Arm.

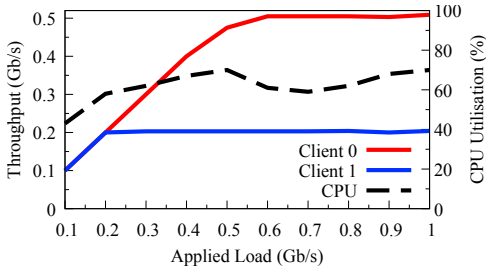

Figure 12: Two client system demonstrating virtualiser swapping and throughput limiting.

Figure 12 shows the system running on the Arm platform. Client 0 has a threshold of 500 Mb/s, and is throttled to 500 Mb/s. Client 1 has a threshold of 200 Mb/s and also throttled to 200 Mb/s. The swap happens at the point where the clients reach 200 Mb/s. The graph clearly shows the bandwidth limitation working as intended. We measure the time to perform the switch to be 17 μs.

6.3.2 Block device

We compare our block device driver and virtualiser with raw reads from Linux on a Samsung SDXC card, see Figure 13. Performance is essentially identical on the two platforms, and limited by the card's performance. Linux does a slightly better job with small block sizes on a random read workload, due to LionsOS presently handling the device sub-optimally. However, LionsOS per-block processing cost is vastly lower than that of Linux.

7 Conclusions

To summarise our experience with building and evaluating LionsOS:

- It is possible to build an OS based on strict separation of concerns, without sacrificing performance. In fact, LionsOS significantly outperforms Linux (and by implication any other modular OSes for which performance data are available).

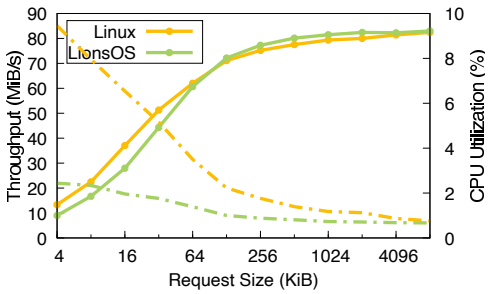

Figure 13: Random read bandwidth on LionsOS and Linux.

- Simplicity wins – it more than compensates for the high context-switch rates resulting from modularity.
- The simplicity of design and implementation makes us confident that the verification aim is achievable. The from-scratch implemented components that make up the system's TCB are all far simpler than other systems verified with automated provers, and similar to the SMT-verified the Microkit.
- The adaptability aim also seems in reach: use-case specific policy modules are simple and easy to swap. Good tooling will be required to help developers design and tailor a system for their use case.
- All this is achieved without changing a single line of code in seL4. In particular, we see no need for shortcuts and compromises such as migrating critical functionality into the kernel, or co-locating services, as Chen et al. [2024] found necessary.

Acknowledgements

LionsOS was made possible through the generous support of multiple organisations: the UAE Technology Innovation Institute (TII) under the *Secure, High-Performance Device Virtualisation for seL4* project, DARPA under prime contract FA 8750-24-9-1000; and NIO USA. Work on the seL4 Microkit and the seL4 device driver framework, which form the foundation on which LionsOS is built, was supported by UK's National Cyber Security Centre (NCSC) under project NSC-1686.

We also thank Ben Leslie from Breakaway for the original Microkit design that led to the LionsOS event-based programming model.

Availability

LionsOS is open source and available at `https://github.com/au-ts/lionsos/`. It has extensive documentation at `https://lionsos.org/`.

References

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, US, October 2003.

James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russell Urzi. A research agenda for mixed-criticality systems, April 2009. URL `http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/`.

Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, US, December 1995.

Per Brinch Hansen. The nucleus of a multiprogramming operating system. *Communications of the ACM*, 13:238–250, 1970.

Buildroot. Buildroot: Making embedded Linux easy, 2016. URL `https://buildroot.org`.

Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys*, 50(6):1–37, 2017.

Can Cebeci, Yonghao Zou, Diyu Zhou, George Candea, and Clément Pit-Claudel. Practical verification of system-software components written in standard C. In *ACM Symposium on Operating Systems Principles*. ACM, November 2024.

Haibo Chen, Shanghai, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, , and Fengwei Xu. Microkernel goes general: Performance and compatibility in the HongMeng production microkernel. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 465–485, Santa Clara, CA, US, July 2024. Usenix.

Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. Atmosphere: Towards practical verified kernels in Rust. In *Workshop on Kernel Isolation, Safety and Verification*. ACM, November 2023.

Darren Cofer, Andrew Gacek, John Backes, Michael Whalen, Lee Pike, Adam Foltzer, Michael Podhradsky, Gerwin Klein, Ihor Kuz, June Andronick, Gernot Heiser, and Douglas Stuart. A formal approach to constructing secure air vehicle software. *IEEE Computer*, 51: 14–23, November 2018.

MicroPython Developers. MicroPython documentation. URL `https://docs.micropython.org/`.

Adam Dunkels. Minimal TCP/IP implementation with proxy support. Technical Report T2001-20, SICS, February 2001. `http://www.sics.se/~adam/thesis.pdf`.

Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In *Workshop on Isolation and Integration in Embedded Systems*, pages 35–40, Glasgow, UK, April 2008. ACM.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys Conference*, pages 177–190, Leuven, BE, April 2006.

Norman Feske. *Genode Foundations*. Genode Labs, 2015. URL `https://genode.org/documentation/genode-foundations/`.

Norman Feske and Christian Helmuth. Design of the Bastei OS architecture. Technical Report TUD-FI06-07-Dezember-2006, Technische Universität Dresden, January 2007. URL `https://www.researchgate.net/profile/Christian-Helmuth/publication/233765173_Design_of_the_Bastei_OS_Architecture/links/0912f50b5c8cd18a4c000000/Design-of-the-Bastei-OS-Architecture.pdf`.

Brett D. Fleisch, Mark Allan A. Co, and Chao Tan. Workplace microkernel and OS: A case study. *Software: Practice and Experience*, 28:569–591, 1998.

Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *ACM Symposium on Operating Systems Principles*, pages 38–51, St Malo, France, October 1997.

Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzyński, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *International Conference on Collaborative Computing*, San Jose, CA, US, December 2005.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review*, 40(3):80–89, July 2006.

Dan Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Microkernels and other Kernel Architectures*, pages 113–126, Seattle, WA, US, April 1992.

Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, US, December 2004.

Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM.

Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. Empirical study towards a leading indicator for cost of formal software verification. In *International Conference on Software Engineering*, page 11, Firenze, Italy, February 2015.

Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and secure inter-process communication for microkernels. In *EuroSys Conference*, Dresden, DE, March 2019. ACM.

Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *USENIX Symposium on Operating Systems Design and Implementation*, November 2020.

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 252–269. ACM, 2017.

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM Symposium on Operating Systems Principles*, pages 225–242, October 2019.

Mathieu Paturel, Isitha Subasinghe, and Gernot Heiser. First steps in verifying the seL4 Core Platform. In *Asia-Pacific Workshop on Systems (APSys)*, Seoul, KR, August 2023. ACM.

Ning Qu. seL4 in software-defined vehicles: Vision, roadmap, and impact at NIO, October 2024. URL `https://sel4.systems/Foundation/Summit/2024/slides/software-defined.pdf`. Keynote at the 6th seL4 Summit.

Qubes. Qubes architecture overview, 2010. URL `https://qubes-os.org/doc/architecture/`.

R.F. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: a system software kernel. *Spring COMPCON*, pages 176–8, 1989.

Freeman L. Rawson, III. Experience with the development of a microkernel-based, multiserver operating system. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 2–7, Cape Cod, MA, US, March 1997.

Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In *ACM Symposium on Operating Systems Principles*, Huntsville, Ont, CA, October 2019. ACM.

Marc Rozier, Vadim Abrossimov, François Armand, I. Boule, Michel Gien, Marc Guillemont, F. Herrmann, Claude Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating systems. *Computing Systems*, 1(4):305–370, 1988.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63:1278–1308, 1975.

seL4 Foundation. The seL4 microkernel, 2021a. URL `https://sel4.systems/`.

seL4 Foundation. Performance, 2021b. URL `https://sel4.systems/About/Performance/`.

seL4 Foundation. seL4 microkit GitHub, 2023. URL `https://github.com/seL4/microkit`.

seL4 Foundation. seL4 project roadmap, 2024. URL `https://docs.sel4.systems/projects/roadmap.html`.

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, Savannah, GA, US, November 2016.

Brent Welch. The file system belongs in the kernel. In *USENIX Mach Workshop*, 1991.

David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2001.

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: A scaleable isolation kernel. In *SIGOPS European Workshop*, pages 9–15, St Emilion, FR, September 2002.

Ian Wienand and Luke Macpherson. ipbench: A framework for distributed network benchmarking. In *Conference for Unix, Linux and Open Source Professionals (AUUG)*, pages 163–170, Melbourne, Australia, September 2004.

Wikipedia. KISS principle, 2001. URL `https://en.wikipedia.org/wiki/KISS_principle`.

Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified NAT. In *ACM Conference on Communications*, August 2017.

Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *ACM Symposium on Operating Systems Principles*, October 2019.