Electronic Design

WILL KEEGAN, Lynx.com

What's the Difference between Separation Kernel Hypervisor and Microkernel?

As the embedded world looks to security solutions to protect connected critical computing functions from external threats, two software platforms have emerged that, at first glance, appear to offer similar functionality. However, under the hood, the approaches and the technologies are quite different.

The Separation Kernel Hypervisor and Microkernel technologies have emerged as the leading contenders in hosting next-generation embedded safety and security critical compute platforms. Both technologies share a great deal in common, stemming from least-privileged design principles, and aim to provide a more robust application runtime environment than traditional monolithic kernel-based OSs. The technologies are similar enough that in the commercial world, the terms are regularly interchanged based on audience or industry requirements, and hence become very confusing for consumers. Despite the similarities in using minimalistic approaches to control CPUs, the kernels are only useful when vendors construct application development platforms on top of them. Once in the hand of the developer, the delivered products can have wild differences at the CPU control level, system assurance properties, development models, and application behavior.

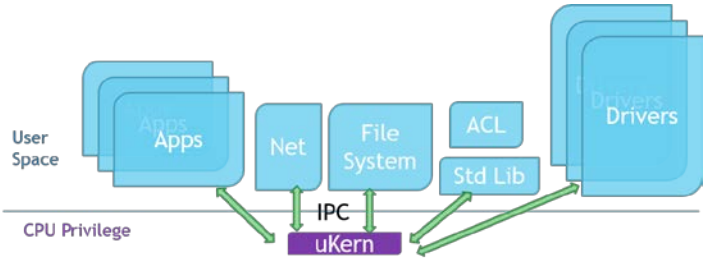
This article embarks on providing an introductory comparison between the two technologies by providing more context of on the origin, expound on their intent to host applications, and highlight some fundamental differentiating properties of the Separation Kernel Hypervisor from Microkernel based platforms.

ORIGIN AND INTENT

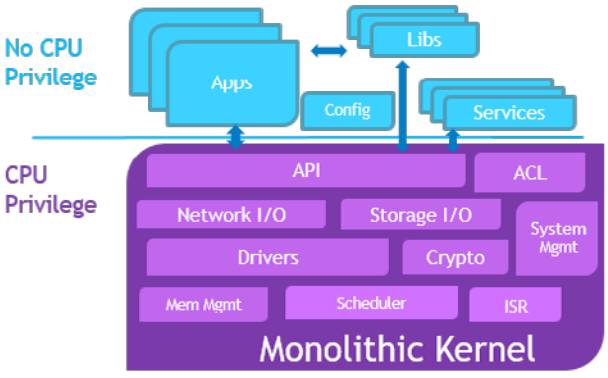
The Separation Kernel Hypervisor and Microkernel concepts have existed for over 30 years with definitions widely available through online and formal publications. The Separation Kernel Hypervisor is an extension of the Separation Kernel originally defined by Dr. John Rushby in his seminal 1981 paper entitled "Design and Verification of Secure Systems" and essentially describes "the task of a separation kernel is to create an environment which is indistinguishable from that provided by a physically distributed system: it must appear as if each regime is a separate, isolated machine and that information can only flow from one machine to another along known external communication lines." The Separation Kernel Hypervisor incorporates the premise and principles of Rushby's Separation Kernel, but imposes an explicit implementation constraint to leverage native CPU virtualization capabilities to autonomously host applications and further enforce separation.

The Microkernel concept comes from many generations of research and development from all over the world, striving to construct efficient CPU control models with the least amount of privileged code necessary to implement an Operating System (*Fig. 1*).

Microkernels aim to provide a safer and more secure run-



1. Here is a diagram of the Microkernel architecture.



2 Above is the Monolithic Kernel architecture.

time environment over the popular monolithic kernel based Operating Systems (*Fig. 2*). Unlike the monolithic kernel approach where all device drivers, I/O, and administrative services run in the same privileged address space to reduce the penalty of context switch time; the Microkernel requires all OS service components to be broken out into separate address spaces at the cost of extra context switching but gaining better integrity protection among internal service components. From an application platform perspective, a Microkernel-based OS and monolithic kernel-based OS look very similar, they are both Operating Systems governed by administrative and configuration policy.

The Separation Kernel Hypervisor, however, aims to support a very different thing—the construction of an n-way fully independent virtually distributed runtime architecture. This architecture demands that there is absolutely no existence of a central controlling Operating System; instead there are a number of operating systems, each one fully independent from the other, and none of which can fully control the physical host.

The Separation Kernel Hypervisor offers a different runtime architecture referred to as a Distributed Heterogeneous Architecture. This independent distributed runtime architecture serves as the main differentiator between the two kernel technologies and has significant advantages over operating systems noted in the following sections.

DIFFERENTIATING PROPERTIES OF THE SEPARATION KERNEL HYPERVISOR

There are many comparable properties that one can examine between a Separation Kernel Hypervisor and Microkernel, like performance, deterministic behavior, trusted codebase, etc. However, these comparisons are only useful if the kernels have the same CPU controlling features and similar application runtime profiles. Imagine comparing the trusted code base of a Separation Kernel Hypervisor capable of scheduling tasks, isolating task memory, and isolating I/O between tasks and physical interfaces, versus a Microkernel that is only capable of scheduling tasks. Obviously, the Microkernel with the lesser of CPU control capabilities will look favorable in a source code line count comparison but it wouldn't be a fair comparison.

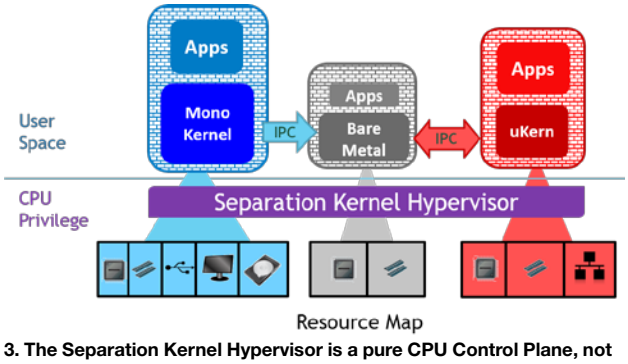
Microkernels have existed in the market much longer than Separation Kernel Hypervisors and have taken many forms from, each one different from the other influenced by the user community and the creative direction of its authors. This introductory comparison does not aim to single out any particular Microkernel implementation. Instead it aims to call out key Separation Kernel Hypervisor properties that have stood out as strong differentiators from Microkernel-based OSs over the years to educate consumers on available capabilities, benefits, properties that they may not have previously been exposed to. Some of the properties noted in these sections are not necessarily be exclusive to Separation Kernel Hypervisors and may over time be incorporated into Microkernel-based platforms.

SCOPE OF KERNEL FUNCTIONALITY

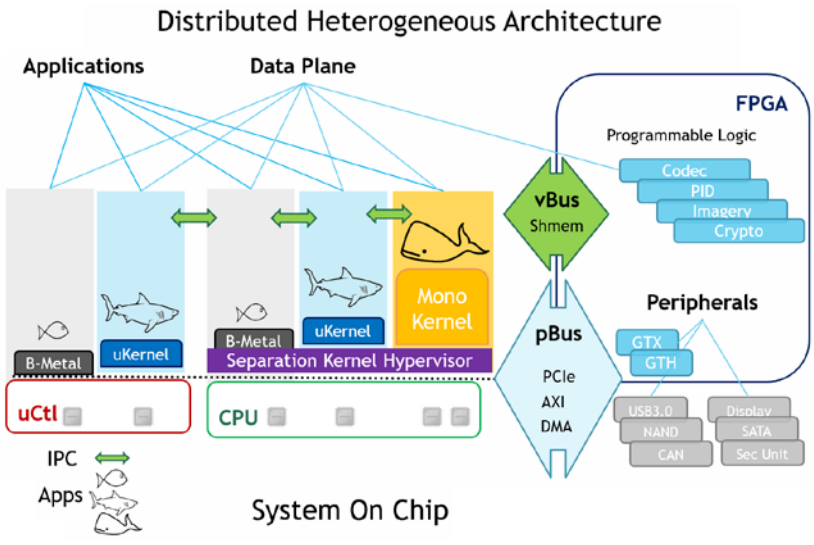
The Separation Kernel Hypervisor is a pure CPU Control Plane, not an application runtime framework like an OS (*Fig. 3*). Its focus is to map raw resources into guest spaces that can be locally controlled by the guest. It is therefore, up to the guest environment to provide a suitable application runtime environment, obviating scope of functionality for the separation kernel hypervisor to provide OS service interfaces as seen in microkernels.

PROVABLE SYSTEM STATE

A driving motivation of Rushby's Separation Kernel invention was to provide a runtime architecture amenable to proving system safety and security properties. The Separation Kernel Hypervisor offers a system configuration model that clearly constrains the runtime architecture of all execution contexts to an explicit tamper-proof set of physical resources. By doing so, evaluators with a discerning eye for safety and security properties can easily argue about the system assurance properties and performance characteristics without depending on a highly complex black-box piece of software. By simply mapping every logical actor in the distributed sys-



3. The Separation Kernel Hypervisor is a pure CPU Control Plane, not an application runtime framework like an OS.



4. Developers are no longer constrained by a single operating system runtime with limited capabilities, and are now freed to mix and match runtime environments.

tem to its physical resource, evaluators can clearly calculate performance metrics or model assurance states by inheriting the immutable physical properties of the hosted logic in question, rather than inheriting assurance properties enforced by a software kernel that has not proven itself enforceable. On modern SoCs with many CPU cores, hosting many applications, a single OS can become a common choke point of vast complexity that can be riddled with security side channels and incomputable deterministic states.

AUTONOMOUS RUNTIME DOMAINS

The autonomy of the application runtime environments is a crucial aspect of a Separation Kernel Hypervisor hosted platform. The autonomy of a guest runtime greatly improves the ability to prove the system states of a software defined system. With autonomous runtimes, software components can be factored out of scope of the proof exercise. If applications were not truly autonomous and were instead dependent on a central resource manager and set of system services, then the scope of the proof exercise must consider the all relationships the central controlling software has with all actors in the system.

Consider a practical system vehicle controller design where two applications need to be separated, e.g., internet-facing application and CAN bus- facing application. On a Separation Kernel Hypervisor, these two applications can run with zero logical links two each other, removing all potential software based exploitations from adversaries to bypass the separation. Even on a least privilege OS, processes are at the mercy of both the integrity of the microkernel and OS system policies, making it difficult to prove that the applications are truly separate. As seen in the infamous Jeep hack (Greenberg, 2015), the attackers used central microkernel-based Operating System services to reverse engineer the system configuration policy to find an exploit that allowed an internet-facing application to control the CAN bus. On a Separation Kernel Hypervisor, the system could have been configured such that the CAN bus interface was never mapped into the internet domain, making CAN bus control a physical impossibility from the perspective of the internet domain.

ISOLATION ENFORCEMENT

Without resource isolation, autonomous runtimes cannot be achieved. The Separation Kernel Hypervisor relies explicitly on recent advancements of CPU memory and I/O controller elements to support hardware assisted CPU virtualization. These advancements under CPU virtualization allow the Separation Kernel Hypervisor to assign portions of hardware resource to guest environment to locally manage without the assistance of the Separation Kernel Hypervisor. These hardware assignments include the ability of guests to independently control their own CPU interface, system memory, and peripheral controllers. On a Separation Kernel Hypervisor, the assignment of resources to guest environments is actually enforced by immutable polices set in the CPU controllers. Having hardware enforce separation provides a tremendous advantage when considering the strength of function, performance efficiency, and strength of evidence in making system assurance property evaluations. This model is starkly different than the service-oriented architecture seen in OS designs where applications would rely on the kernel to manage memory, CPU execution contexts, and broker connectivity to I/O. Many of these OS models rely solely on software based policies that could be potentially tampered with by a software actor that gains CPU or OS administrative privilege.

DISTRIBUTED HETEROGENEOUS RUNTIME ARCHITECTURE

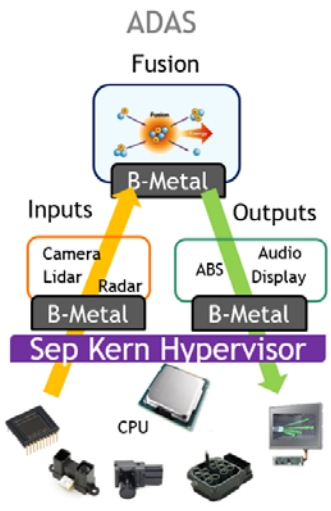
The distributed heterogeneous runtime architecture of the Separation Kernel Hypervisor is the most distinguishing characteristic from the centralized homogenous architecture commonly seen in in a Microkernel-based OS.

Building upon the resource isolation, and autonomous runtime domain properties previously described, the Separation Kernel Hypervisor adds in the ability to host a variety independent Operating Systems through CPU virtualization, including Microkernel based OSs. Under this architecture, each independent OS locally manages their own virtual machine

contained with a single CPU context, thus implementing Rushby's vison to "create an environment which is indistinguishable from that provided by a physically distributed system."

In the advent of next generation multicore SoCs, the Separation Kernel Hypervisor can provide the ability to map in all aspects of theses SoC and independently control portions with operating systems and development languages that best suites the need for the target environment. Developers are no longer constrained by a single operating system runtime with limited capabilities, and are now freed to mix and match runtime environments (*Fig. 4*).

It is important to note that both Separation Kernel Hypervisor and Microkernel run-time models feature a least-privilege design, but the Separation Kernel Hypervisor model opens the door in providing greater evidence of assurance properties and offers tremendous flexibility for end users to construct highly advanced systems where software can take the most advantage out the underlying resources with minimal development efforts.



5. With the bare-metal context, applications can run in a completely raw CPU and flat memory context with zero interference from any other application or kernel services.

BARE-METAL RUNTIME

One of the most interesting runtime profiles the Separation Kernel Hypervisor offers for the embedded space is the 'bare-metal' runtime to accommodate a vast set of applications commonly seen in microcontroller designs that do not require the support of the services of an OS. Consider a sensor fusion application that simply requires polling on memory mapped I/O interfaces (*Fig. 5*). With the bare-metal context, these applications can run in a completely raw CPU and flat memory context with zero interference from any other application or kernel services.

SUMMARY

Comparing the two classes of technologies is indeed a complicated task. In many respects the Separation Kernel Hypervisor and Microkernels are very similar. Both have minimal code bases, and both can support least privilege architectures. However, looking in more closely, the two technologies aim to be different things, and their least-privilege runtime architectures can be constructed in different ways.

Where Microkernels aimed to provide a safer runtime environment over monolithic kernel based OSs, the Separation Kernel Hypervisor aims to be something different – to not be an operating system. This entails having no centralized system control, and providing a variety of application development and portability options, and offering a runtime architecture that can support strong assurance claims through the utilization of modern CPU virtualization capabilities.