KOS - Principles, Design, and Implementation

Martin Karsten, University of Waterloo

Work In Progress - September 29, 2015

Abstract

KOS is an experimental operating system kernel that is designed to be simple and accessible to serve as a platform for research, experimentation, and teaching. The overall focus of this project is on system-level infrastructure software, in particular runtime systems.

1 Introduction

KOS (pronounce "Chaos") is an experimental operating system kernel that is designed to be simple and accessible to serve as a platform for research, experimentation, and teaching. It is focused on building a nucleus kernel that provides the basic set of operating system functionality, primarily for resource mediation, that must realistically be implemented for execution in privileged mode. While being simple KOS is not simplistic and avoids taking design shortcuts that would prohibit adding more sophisticated resource management strategies later on – inside the kernel or at higher software layers. The nucleus kernel is augmented with several prototype subsystems typically found in an operating system to eventually support running realistic applications. The entire code base is written in C++, except for small assembler parts. The C++ code makes use of advanced language features, such as code reuse with efficient strong type safety using templates, the C++ library for data structure reuse, as well as (limited) polymorphism. Existing open-source software is reused for non-nucleus subsystems as much as possible. The project is hosted at `https://git.uwaterloo.ca/mkarsten/KOS`

2 Motivation

In principle, an operating system has two basic functions. First, it consolidates low-level hardware interfaces and provides higher-level software abstractions to facilitate and alleviate application programming. Second, as a multiprocessing system, it manages all system resources and mediates between possibly conflicting demands of multiple applications that concurrently use a computer system. This also includes isolating application state from each other. The operating system kernel is the core software component executing in privileged mode and

must at least implement a basic set of functionality to mediate access to system resources. However, in contrast to a virtual machine hypervisor, an operating system kernel typically supports dynamic resource allocation and also facilitates collaboration between different applications.

The classical *monolithic* approach to building an operating system kernel places all code and state into a single addressing space and uses subroutine calls as the main form of modularization. In contrast, a *micro-kernel* is comprised of a much smaller kernel executing in privileged mode, while providing higher-level programming abstractions and resource policies through separate processes that communicate with the kernel using message passing. At the other end of the design spectrum is the *exokernel* design in which a minimalistic kernel only provides a low-level interface exposing much of the underlying hardware, while applications are built using libraries that provide higher-level abstractions based on this low-level interface.

The overarching design principle for KOS is simplicity without being simplistic[1]. In terms of the kernel architecture, this principle translates into a minimal but operational kernel design. The kernel provides all necessary functionality to run applications, but relegates complex resource policies to other parts of the software stack. In particular, the same kernel nucleus can be used as a building block for a monolithic operating system, but could also be deployed in a micro-kernel or exokernel setup. It can form the basis for a hypervisor or a unikernel. This design concept is termed *varikernel* here. Using a metaphor from biology, a varikernel is similar to a *pluripotent* stem cell and can differentiate into different specialized kernels. In fact, it seems entirely feasible to make different design decisions regarding a monolithic, micro-kernel, or exokernel setup for each subsystem that is added to the kernel nucleus.

The motivation for simplicity is threefold. First, an operating system should primarily get out of the way of applications striving for best performance. Second, complex kernels are anathema to highly controlled experiments aimed at systematically understanding and documenting design and performance trade-offs in the design of system software and services [17]. Finally, meaningful consolidation of scientific-level knowledge about systems programming, along with the necessary independent validation of research claims, can only be attained with a compact reference platform. Ultimately, curating knowledge is an important scientific task, especially as it pertains to academic teaching.

Based on this motivation, the primary design objective is building a kernel nucleus that can operate with arbitrarily many cores and massive amounts of memory without expending unsustainable resources when used for smaller-scale systems. Most importantly, the minimalistic kernel should not impede the efficient execution of applications or adding features or complex subsystems later on. Furthermore, a network stack is given priority over supporting a local hard disk and file system. With recent developments in network speeds, memory prices, and non-volatile random-access memory (NVRAM), it seems less important to ingrain the dependency on a hard disk or traditional file system deep into

[1] http://www.presentationzen.com/presentationzen/2009/06/simplicity-in-las-vegas.html

an operating system kernel than it used to be. One particular idea is for KOS to ultimately provide a somewhat uniform interface across distinct memory domains. For multi-core systems, per-core state can be viewed as a partial cache of the global system state, which supports the notion of dynamically adding and removing cores. This principle can be expanded to distributed memory by avoiding any dependencies on local persistent memory.

KOS is designed to embody a future-proof structure for a kernel nucleus that is functionally scalable from small devices to supercomputers. Existing production-level operating system kernels cannot satisfy these objectives, because they are encumbered by legacy, while research kernels are typically built to showcase a particular design idea. On a philosophical level, KOS is coded with readability and accessibility in mind, not least to support teaching. It is intended to expand the notion of "open source" for systems-level software by making it as understandable as possible – without compromising its functional ability and performance.

3 Design

KOS uses a moderate object-oriented design with key components of the system represented by classes that can be found in respective source files. These are *Machine*, *Paging*, *Processor*, *FrameManager*, *AddressSpace*, *Process*, *Thread*, *Scheduler*. These components are further discussed in subsequent parts of this document. The most mature components that provide interesting characteristics are the thread runtime (Sec 4.5) and memory management (Sec 4.6) components. Both are designed to be compact yet extensible and reflect the overall design philosophy of KOS.

3.1 Source Code

The source code is split into the following directories under `src/`, listed here roughly in order of increasing dependencies on other parts of the code base.

- `include` - header files that are used in both kernel and user-level
- `generic` - data structures and functionality that is independent of other parts of the code base
- `runtime` - thread runtime, including scheduler and blocking synchronization mechanisms; platform dependencies confined to namespace *Runtime*
- `kernel` - machine-independent code, including memory and process management
- `extern` - third-party software components
- `main` - services and tests started after kernel bootstrap
- `world` - communication and storage facilities

- `machine` - machine-dependent code, including paging
- `devices` - bus support, device drivers
- `gdb` - gdb stub/server for remote debugging
- `ulib` - user-level library
- `user` - user-level programs

The system is roughly structured into three layers: *machine*, *kernel*, and *runtime*. The *machine* layer implements mechanisms that are specific to a particular hardware platform, currently only `x86-64`. The central *kernel* layer implements KOS-specific functionality, while *runtime* is designed to be independent of KOS, so that it could also be used for a user-level threading runtime system.

3.2 Communication

A central design decision for an operating system kernel is the choice of communication mechanism used between system components. Two dimensions characterize a communication mechanism: synchronization and parameter passing. If the requested service is executed synchronously, the interaction pattern is typically easier to understand and use by programmers than asynchronous execution. With entities operating in different address spaces or under different protection regimes, typically call-by-value is used for parameter passing, such as for classical system calls. Otherwise it is possible to use call-by-reference.

Synchronous call-by-value between address spaces incurs direct overhead, which traditionally has been considered as the primary overhead of a microkernel setup [18]. For the special case of user/kernel system calls, this direct overhead is greatly reduced by embedding the kernel address space in each user address space. The direct cost of the privilege change associated with a system call has been reduced significantly with the introduction of dedicated machine instructions, such as `syscall` or `sysenter` [13]. However, synchronous execution of disjoint software components can lead to cache distortion. Asynchronous execution requires intermediate buffering, but can result in better cache utilization [21]. Furthermore, asynchronous call-by-value is the only mechanism available in a distributed system, while asynchronous call-by-reference in shared-memory configurations requires agreement about the ownership (memory allocation, write access) of parameters.

Recent literature about research operating systems argues for a strict share-nothing approach between kernel subsystems, because it is claimed that an increasing number of cores will make cache coherency infeasible [15, 22]. Then again, Martin et al. [19] claim that cores on the same chip will always have access to efficient coherency mechanisms that are necessary to provide a shared memory hardware interface. One of the research goals for KOS is searching for an ideal set of basic communication mechanisms that provide efficiency and flexibility for a variety of hardware configurations. Currently, whenever possible, asynchronous execution is the default interaction pattern.

KOS supports shared-memory for communication and thus call-by-reference semantics with lower overhead. However, a number of services are implemented using asynchronous execution to reduce cache distortion, benefit from batching, and to avoid the immediate waiting overhead of synchronous execution. Furthermore, the diskless design of the kernel nucleus will make it easier to run multiple (or different) kernel instances on different cores in a multi-core system and enable support for multikernel [15] and satellite kernel [20] scenarios. A multiprocessing system on parallel hardware provides benefits for two generic scenarios: *consolidation*, i.e., running multiple independent services, and *parallelization*, i.e., running a collection of dependent operations faster. Parallelization is limited by the effects described by Amdahl's Law [14], but consolidation is typically fairly scalable. Support for shared-memory and message-passing is aimed at supporting both consolidation and parallelization well.

4 Details

4.1 Tool Chain

KOS is built using a custom tool chain comprised of `gcc`, `binutils`, and `newlib` [11]. `newlib` is a compact C library that is well integrated with `gcc`, thus suitable for bootstrapping a tool chain. It is used inside the kernel to provide basic standard C utility routines in support of `gcc`'s C++ library. It is currently also used as the C library for user-level programs. Using this custom tool chain and its libraries, `clang++` can also be used to compile KOS.

4.2 Software Reuse

Aside from the C++ standard libray, the code base currently reuses the following third-party software packages. Care is taken to not ingrain external packages to deeply into system, but instead ensuring a strong separation via APIs and glue layers, such that external software packages can be upgraded with relative ease.

- The ACPI Component Architecture [1] is used to identify hardware resources during bootstrap.
- A network driver (e1000) from the Common Driver Interface project [4] is used for network connectivity.
- Doug Lea's malloc [5] memory allocator is used for flexible memory allocation, e.g., in support of the C++ standard library.
- The ELFIO [7] library is used to load ELF binaries for execution.
- The lwIP [10] stack is used as network stack.
- The Multiboot2 specification from the `grub` project [9] is used to retrieve system information from the bootloader.

4.3 Bootstrap

The system boots using grub. The multiboot header and bootstrap routine is implemented in `src/machine/boot.S`. After switching the *bootstrap processor* (BSP) into 64-bit mode and enabling paging (using identity mapping), control is transferred to `kmain()` in `src/kern/Kernel.cc`. The same code path is used later to boot additional processors, termed *application processor* (AP), but differentiated in `kmain()`. For the BSP, `kmain()` invokes `Machine::initBSP()`, while for APs, it invokes `Machine::initAP()`. `Machine::initBSP()` performs the basic bootstrap, including paging, and then invokes `Machine::bootMain()`. After further initialization in `Machine::initBSP2()`, which includes starting up APs, the main BSP thread executes `kosMain()`, while APs initially execute the idle loop `Runtime::idleLoop()`.

4.4 Debugging

KOS supports `gdb`'s remote debugging mode using a serial interface. The debug support is available after the initial bootstrap phase and connects to the debugger at the beginning of `Machine::initBSP2()`. Due to limitations in the timer devices and interrupt handling in KOS, `gdb` can currently only be used in `all-stop` multithreading mode. Debug and logging output is sent to a serial interface, as well as `qemu`'s debugging port, if available.

4.5 Thread Runtime

The thread runtime system is designed to be simple, but extensible. Each per-core *Scheduler* object implements a ready queue with a small number of priority levels. Threads are implemented as a class of the same name and typically resumed to the same scheduler, but there is also a simple load-balancing mechanism executed during preemption (`Scheduler::preempt()`) that can move threads between schedulers. It is possible to set thread affinity to a particular scheduler, i.e., core.

The thread context-switch is implemented as a standard context-switch saving and restoring caller-owned registers. In addition, if a context-switch suspends or resumes a user thread, the `fs`/`gs` registers are also saved and restored respectively. The appropriate floating point control bits in the `CR4` register are enabled, but no floating point context-switch is implemented yet.

Synchronization

Standard blocking synchronization primitives, such as mutex lock, semaphore, and condition variable, are implemented in a generic way. Blocking wait operations can be given a timeout interval. Asynchronous cancellation of a blocking operation introduces a fundamental race condition with the actual event signalling, which is especially perilous, because most thread queues in KOS are implemented as intrusive containers. KOS provides a compact solution to this problem. A thread entering a blocking wait sets up a polymorphic object on the

stack that describes the nature of the blocking operation, as well as the a possible timeout, and stores a pointer to this object in the thread's metadata. When resuming a thread, an atomic operation is used to read and reset this pointer, which determines a unique "winner" between multiple asynchronous attempts to unblock the thread (`Thread::unblock()`). The winner proceeds to resume the thread and cancels the respective other wait indication. This functionality is implemented in `runtime/BlockingSync.h`.

The KOS thread runtime is designed for dual-use as both kernel-level and user-level threading system. The research question is whether such a design is feasible and useful. To this end, all platform-dependent runtime code is sequestered in namespace *Runtime* implemented in files `runtime/Runtime.h` and `runtime/RuntimeImpl.h`.

Scheduling

Other research questions are concerned with preemption aggressiveness and scheduling. Thread preemption is fundamentally a protection mechanism ensuring fair access to the computing resource. However, in a system that is not running at full load, it is not necessary to rely on preemption to ensure adequate progress of all threads. Thus, preemption becomes only relevant when the system is fully loaded. On the other hand, preemption can be used to guarantee an upper bound on responsiveness. For that purpose, it is also useful to consider thread priorities. Given the inherent overhead of preemption, the best balance between fairness and undisturbed execution is unclear. This question is closely related to scheduling policy. A scheduler can be designed for strict fairness and load balancing, but this might be harmful for the overall system throughput, especially when considering scalability to a large number of cores. In contrast, thread blocking and resuming are operations that are used frequently for I/O-oriented workloads – regardless of the overall load situation. In summary, the trade-off between optimizing for throughput vs. responsiveness constitutes an important design question for a scheduler.

4.6 Memory Management

Physical Memory

Phyiscal memory is managed in class *FrameManager*. Multiple *FrameManager* instances can represent multiple contiguous areas of memory, for example in a NUMA system, but currently only a single instance is used. KOS supports multiple page/frame sizes and organizes physical memory accordingly. For each frame size, a fixed-size hierarchical bitmap is used to indicate the availability of each memory frame. A hierarchical bitmap can be searched efficiently using bit-string instructions that are available on most contemporary hardware platforms. Searching for contiguous memory is slightly less efficient, but it is assumed that contiguous physical memory spanning multiple frames is needed only rarely - typically only in the context of (legacy) device initialization. The memory

cost of representing all physical memory in this way is miniscule ($\sim 0.006\%$ of the physical address range) and it allows managing physical memory using a fixed-size, static data structure, i.e., without a circular dependency on dynamic memory allocation. The *FrameManager* class currently uses a single lock per frame size to protect internal data structures during concurrent requests. KOS contains a proof-of-concept implementation of asynchronous memory wiping (zeroing) during the idle loop, which is also implemented using a fixed-size hierarchical bitmap per frame size.

Paging

The basic mechanisms for paging are implemented in class *Paging*. KOS uses recursive page tables and embeds kernel memory in user address spaces. The mapping and unmapping between physical frames and virtual pages is done using atomic operations without explicit locking. The paging code is written using integer templates to express similar functionality at different page table levels. This results in clean and compact source code while ensuring efficient executable code due to forced unrolling of non-tail-recursive routines.

Virtual Memory

Class *AddressSpace* represents and manages a virtual memory context, which becomes manifest in a page table hierarchy rooted in the `CR3` register. A dedicated global object termed `kernelSpace` represents kernel memory independent of any user address space. The kernel internally uses 2MB pages for heap memory. Aside from the hardware page tables, KOS does not maintain a separate data structure containing information about existing page mappings. The available range of virtual memory is managed using a simple single marker separating unused from used virtual memory, protected by a lock. For special mappings, such as shared memory or swapped pages, it is planned to create dedicated data structures. For example, shared memory can possibly be managed by a global inverted page table indexed by physical address. For simplicity, intermediate page tables that are not needed anymore are not automatically released in the current version of KOS. The resulting memory overhead is $\sim 0.2\%$ of the maximum virtual memory range used in an address space. All page tables are released when an address space is destroyed.

With respect to virtual memory management, KOS takes a different and more laissez-faire approach than some of the recent literature, in line with its stated goals of simplicity and minimalism. The simple VM management strategy embodied in KOS is probably sufficient for most small and medium-sized applications without burdening the kernel with complex mechanisms and policies. However, it might lead to significant and non-recoverable fragmentation, if a user-level application causes detrimental memory mapping patterns. Then again, with a proper memory allocator implemented in a system library, such paging patterns should be very rare and considered a application deficiency. The virtual memory region in an address space is not a system resource, therefore a

deficient application primarily harms itself.

Similarly, a concurrent application might issue many concurrent mapping requests, which proceed in strict serial order in KOS, because of the single lock protecting the virtual address space. Existing literature, such as [16], describes this as a challenge that needs to be addressed by the operating system kernel. While this might be sound engineering from a production system perspective, fundamentally, fixing higher-level deficiencies in lower-level software burdens the common lower layer with extra complexity and overhead. A research question for KOS is verifying the conjecture that such a request pattern does not need to be directly handled by the kernel. Instead, the memory allocator in the system library can mitigate frequent concurrent memory allocation requests and bundle memory management, such that it results in only infrequent mapping requests to the kernel.

Asynchronous TLB Invalidation

An experimental feature in KOS is asynchronous TLB invalidation in lieu of the synchronous, and typically costly, TLB shootdown. When memory is unmapped, both virtual and physical addresses are not directly reusable, but the memory area is stored in a per-address-space list of dangling mappings, along with the number of processor cores that currently execute in this address space. Because an address space switch invalidates TLB entries anyway, the dangling memory list is updated during each such switch and keeps track of how many processor cores might have TLB entries pointing to each area. When the count reaches zero for an area, both virtual and physical address ranges are released for eventual reuse. The asynchronous TLB invalidation scheme is designed, such that its algorithmic complexity is strictly linear in the number of memory ranges. Kernel pages are mapped with the `G` paging bit and not automatically invalidated during an address space switch. Thus, for kernel memory, dangling TLB entries are explicitly invalidated during each address space switch. Furthermore, this explicit page invalidation is run during each preemption interrupt for both kernel and current user address space, regardless of an address space switch. This works well for page unmapping, but does not directly maintain the synchronous semantics of the POSIX `munmap` system call. Further work is needed to add synchronous semantics, i.e., make `munmap` effectively a blocking system call. Also, TLB invalidation is required in other scenarios, such as memory locking or page swapping, which are not supported yet.

4.7 Process Management

The *Process* class inherits from *AddressSpace* and represents a traditional user-level application process. A thread in a process is encoded as an object of class *UserThread*, which inherits from *Thread* and contains a user-level stack in addition to the kernel stack. Class *Process* provides ELF binary loading and thread management. It also holds references to I/O objects available to a process. The design of process management is not finalized and the current

implementation simply represents an attempt to provide a subset of POSIX-like functionality towards running real-world applications.

4.8 Coding Details

System-level software often uses the `__builtin_expect()` function to provide the compiler with branch prediction information. For example, in the Linux kernel macros `likely()` and `unlikely()` are used to express whether a conditional is likely to evaluate true or not. However, the term 'likely' and 'unlikely' carry semantics that naturally lend themselves to optimizing for the common case. In contrast, when the system is fully loaded, conditions might evaluate to the unlikely case, yet it would be particularly beneficial to execute the fast path under those conditions. Therefore, KOS code uses macros `fastpath()` and `slowpath()` to express more explicit semantics and, where applicable, branch prediction is optimized for the high-load case, rather than the common case.

4.9 User vs. Kernel

KOS contains experimental support for a `privilege()` system call, which allows executing user-level code in privileged mode. Using this system call, trusted applications or services can execute mostly in user mode, but occasionally perform privileged operations without having to explicitly communicate with the kernel. This should help with building micro-kernel scenarios and/or support user-level runtime systems that, for example, would benefit from being able to disable system-level preemption for short periods of time.

4.10 Interrupt Handling and Device Drivers

Interrupt handling is somewhat ad-hoc and not very mature yet. Most hardware interrupts are handled asynchronously by storing the event in a bitmap. A dedicated interrupt handling thread executes asynchronous interrupt handlers in response to bits being set in this interrupt bitmask. Further, there is no comprehensive framework for device drivers yet. KOS supports a PS/2 keyboard, serial devices, the PIT and RTC timer devices, as well as a small set of network cards through CDI's e1000 driver [4]. In particular, APIC timers are not yet used for preemption, but instead per-core preemption is emulated by sending IPIs based on the periodic RTC interrupt. Also, interrupt priority levels are not yet used, but instead interrupts are either completed enabled or disabled.

5 Next Steps

Aside from removing the limitations and working towards the research objectives outlined above, the major next steps for the project are the following:

- implementing a system-wide rendezvous service, including access control

- investigating other open-source projects, such as Genode [8] and Rump Kernels [12], and re-using a more comprehensive C-library, as well as device drivers
- integrating a simple diskless file system, such as BFS [2]
- maturing the system to being able to run simple applications, such as a simple shell, BusyBox [3] or Dropbear SSH [6]

6 Related Work

Aside from well-known open-source production-level operating systems, such as Linux and the *BSD variants, there are commercial systems, such AIX, HP-UX, QNX, Solaris, Windows, etc. The following list gives an overview of current open source research operating system kernel projects in alphabetical order. The given descriptions are directly taken from each project's web page. None of them has the same objectives as KOS or seems directly transformable to satisfy theses objectives. **TODO**: details, discussion

- `http://akaros.cs.berkeley.edu` – *Akaros* is an open source, GPL-licensed operating system for manycore architectures.
- `https://arrakis.cs.washington.edu` – *Arrakis* is a new operating system that is designed around recent application and hardware trends: Applications are becoming so complex that they are miniature operating systems in their own right and are hampered by the existing OS protection model.
- `http://www.barrelfish.org` – *Barrelfish* is a new research operating system being built from scratch and released by ETH Zurich in Switzerland, with assistance from Microsoft Research.
- `http://www.contiki-os.org` – *Contiki* is an open source operating system for the Internet of Things.
- `http://groups.csail.mit.edu/carbon/?page_id=39` – *Factored Operating System* is a new operating system targeting multicore, manycore, and cloud computing systems with scalability as the primary design constraint, where space sharing replaces time sharing to increase scalability. – No code available.
- `http://femtoos.org` – The *Femto OS* is a very concise portable real time - preemptive operating system (RTOS) for embedded microcontrollers with minimal ram and flash.
- `https://www.haiku-os.org` – *Haiku* is an open-source operating system that specifically targets personal computing.

- http://harvey-os.org – *Harvey* is an effort to get the Plan 9 code working with gcc and clang.
- https://en.wikipedia.org/wiki/L4_microkernel_family – *L4* is a family of second-generation microkernels, generally used to implement Unix-like operating systems, but also used in a variety of other systems.
- http://www.minix3.org – *MINIX 3* is a free, open-source, operating system designed to be highly reliable, flexible, and secure.
- http://muen.codelabs.ch – The *Muen* Separation Kernel is the world's first Open Source microkernel that has been formally proven to contain no runtime errors at the source code level.
- https://github.com/aclements/sv6 – *sv6* is a POSIX-like research operating system designed for multicore scalability based on *xv6*.
- http://www.syllable.org – *Syllable* is becoming an easy-to-use free software operating system for the home and small office user.
- http://visopsys.org – *Visopsys* is an alternative operating system for PC compatible computers.
- https://github.com/mit-pdos/xv6-public – *xv6* is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6.

7 Conclusion

KOS is work in progress. After the initial development phase, certain objectives for simplicity have been achieved – for example in the thread runtime and memory management subsystem. The code is compact and accessible. However, significant further work is necessary to fully investigate the conjectures underlying this work.

References

[1] ACPI Component Architecture. https://www.acpica.org.

[2] Behrooz File System. http://bshafiee.github.io/BFS.

[3] BusyBox. http://www.busybox.net.

[4] Common Driver Interface. http://www.lowlevel.eu/wiki/Common_Driver_Interface.

[5] Doug Lea's malloc. http://g.oswego.edu/dl/html/malloc.html.

[6] Dropbear SSH. https://matt.ucc.asn.au/dropbear/dropbear.html.

[7] ELFIO. http://elfio.sourceforge.net.

[8] Genode Operating System Framework. http://genode.org.

[9] GNU GRUB. https://www.gnu.org/software/grub.

[10] lwIP - A Lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip.

[11] Newlib. http://sourceware.org/newlib.

[12] Rump Kernels. http://rumpkernel.org.

[13] SYSENTER/SYSEXIT vs. SYSCALL/SYSRET. http://newsgroups.derkeiler.com/Archive/Comp/comp.arch/2010-01/msg00215.html.

[14] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of AFIPS (Spring)*, pages 483–485, 1967.

[15] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of SOSP*, pages 29–44, 2009.

[16] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of EuroSys*, pages 211–224, 2013.

[17] Ashif S. Harji, Peter A. Buhr, and Tim Brecht. Our Troubles with Linux Kernel Upgrades and Why You Should Care. *SIGOPS Oper. Syst. Rev.*, 47(2):66–72, July 2013.

[18] Jochen Liedtke. Improving IPC by Kernel Design. *SIGOPS Oper. Syst. Rev.*, 27(5):175–188, December 1993.

[19] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Commun. ACM*, 55(7):78–89, July 2012.

[20] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of SOSP*, pages 221–234, 2009.

[21] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of OSDI*, pages 1–8, 2010. https://www.usenix.org/conference/osdi10/flexsc-flexible-system-call-scheduling-exception-less-system-calls.

[22] David Wentzlaff and Anant Agarwal. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.