The Hierarchical Microkernel: A Flexible and Robust OS Architecture

Stefan Winter, Martin Tsarev, Dennis Feldmann, Robert Reinecke

DEEDS Group
TU Darmstadt, Germany
moduli-os@deeds.informatik.tu-darmstadt.de

Abstract

The diversity of applications, hardware platforms and needs for features, e.g. in terms of performance and security, necessitate the continual development of either customized operating systems (OSs) or modifications of existing ones. This is daunting from the efforts for continual re-design and integrity assurance of each OS.

In order to provide high-integrity, robust by-design and adaptable OSs to match evolving application-driven computing environments, we propose a highly customizable and robust OS architecture (the Hierarchical Microkernel – HM). The proposed HM architecture enables the *flexible* design and implementation of highly customizable (at *design time*) and highly adaptable (at *run-time*) robust OSs.

I. Introduction

Mediating across user application requirements and the rapidly changing hardware capabilities, OSs are required to adapt if either changes. On the hardware side, a number of recent developments are not reflected by modern OSs: "shared-nothing" systems are on the rise [1], GPGPUs and FPGAs are expected to become standard building blocks of computer systems [2], and nano-scale processors will require software-based error recovery due to increased hardware failure rates [3].

At the same time, OSs need to provide guarantees and mechanisms to meet application integrity requirements, especially if physical machines are shared among several users running applications of differing criticality. Virtualization partially solves such issues but leads to unnecessarily increased memory and CPU overhead, as OSs are fully replicated within virtual machines.

Classical OS architectures usually match a limited spread of hardware architectures and application requirements (i.e. constrain *design flexibility*) and tolerate limited changes in their execution environments (i.e. constrain *run-time flexibility*). Monolithic OSs, for example, typically lack run-time flexibility, as large parts of the system are statically linked into a monolithic binary where individual parts are difficult to exchange at run-time. While classical microkernels support reconfiguration, they lack run-time flexibility if they contain either large monolithic servers or servers that cannot be exchanged individually during operation. However, if microkernel systems are constructed from small insular components to avoid this constraint, the microkernel as a central coordinating instance becomes a performance and scalability bottleneck.

The proposed *hierarchical microkernel* (HM) is a novel OS architecture with flexibility and robustness as its primary design goals. *Flexibility* being "the ease with which a system or component can be modified to use in applications or environments other than those for which it was specifically designed" [4] covers both design and run-time aspects. *Robustness* is the ease with which integrity requirements are maintained. We achieve flexibility and robustness by the following design choices.

1) Fine-grained system composition from tiny components, called *modules*
2) Inter-module communication solely based on *message-passing* for loose coupling and a *local broadcast* communication paradigm across neighboring modules, enabling the safe replacement of components at run-time: component replicas can eavesdrop on neighbors they are expected to replace
3) *Hierarchical organization*: Modules are organized in a mono-hierarchy, where each module is directly responsible for managing its subordinate child modules
4) The parent-child relationship fosters an asymmetric *trust* relation similar to the supervisor-/user-mode boundary found in many OSs: We require children to rely on their parents but not vice versa.

II. Related Work

While individual dedicated robust/trustworthy OSs such as Singularity [5] exist, general architectures for flexible and robust OSs are lacking. OSs that are constructed using modular programming language concepts possess design flexibility to a certain degree. In K42 [6], for example, resources are represented by objects in the OS to leverage the modularity of the object-oriented programming paradigm. However, K42 targets a specific hardware platform and is not designed for portability to very diverse platforms.

Library OSs [7] do offer this design flexibility if they provide functionality in sufficiently fine-grained library implementations. However, as they do not inherently support

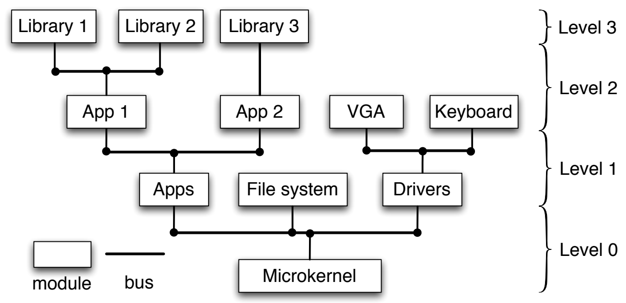

Figure 1. Example of a HMOS

loose coupling between applications and the system components (libraries) they are linked with, there can be strong interdependencies that complicate the implementation of interposition mechanisms, hot-swapping [8] or dynamic updates [9], thus hampering run-time flexibility.

HM intends to provide both design and run-time flexibility, thereby enabling adaptation to both architectural changes (by enabling module reuse and selective re-implementation) and application requirements at run-time (by facilitating hot-swapping of modules).

In multi-core and multiprocessor OSs message-passing is applied to reflect shared-nothing architectures and to increase component isolation for both flexibility and robustness [10], [11]. Hence, we expect HM-based OSs (HMOSs) to perform competitively on multiprocessor systems.

III. HM Architecture

The HM is a composition of two core elements – modules and buses as depicted in Figure 1. *Modules* are self-contained functional entities, resembling both active or passive elements (e.g. processes and libraries) of commodity OSs. Modules are isolated from each other similar to processes in traditional OS architectures. Interaction between modules is accomplished solely using message-passing over communication channels called *buses*. Modules use a simple send/receive interface for exchanging messages over a bus.

A. Inter-module communication

Unlike traditional inter-process communication (IPC) paradigms where two components create virtual point-to-point channels, buses can be shared by an arbitrary number of modules and messages are always locally broadcasted to all attached modules. The implementation of point-to-point communication on top of broadcast is still possible if this is required by the attached modules, just as in Ethernet network communication. Broadcast message-passing has the advantage that a sender does not need to know (and specify) details of the system organization, e.g. the location of the recipient, to initiate communication. Broadcast communication also enables the safe replacement and recovery of components at run-time. For example, a monitoring module can journal all messages on the bus and keep track of the attached (child) modules' states. In case of a module failure, the monitor can aid detection, initiate a restart of the module, and recover its previous state from the monitored message sequence.

Message-passing structurally turns OSs into inherently distributed systems with all their benefits in terms of scalability [12], [11]. Buses within the system can be chosen or implemented according to the requirements of the system designer or attached modules. It is possible to apply industry standards (e.g. MPI, IP, etc) for compatibility with existing systems and communication networks.

B. HM's hierarchical organization

To avoid centralized resource management, which could become a performance bottleneck or a single point of failure, our architecture follows the divide-and-conquer principle to distribute the abstraction and management of system resources.

To achieve this, modules are composed hierarchically, leading to several distinct *levels* in the system (cf. Figure 1). The number of levels in the system depends on the OS designer's choice of the hierarchy depth. The microkernel is the root module and provides a basic hardware abstraction to a set of modules over a bus. As this bus is attached "on top" of the microkernel, it is referred to as the microkernel's *child bus*. The microkernel and its child bus are operating on Level 0. The microkernel's child bus is referred to as *Bus 0*. The other modules connected to Bus 0 are called Level 1 modules. They can provide abstractions to Level 2 modules using separate Level 1 buses.

Except for the microkernel, which does not have a parent bus, each module has to be attached to exactly one parent bus and each bus has to be attached to exactly one parent module. This results in a mono-hierarchical organization visualized by a tree structure where nodes represent modules and edges between them represent buses. In Figure 1, Bus 0 is the parent bus of the modules *Apps*, *File system* and *Drivers*. The microkernel, as the parent module of Bus 0, is referred to as the parent module of these child modules. Parent modules are responsible for managing the resources of their direct children. Hence, they can apply management algorithms and policies, e.g. for scheduling or resource allocations, that suit their requirements best.

Modules that are connected to a parent bus and a child bus are called *gateways*, as they can forward or translate messages between these buses. In Figure 1 the modules *Apps*, *Drivers*, *App1* and *App2* are gateways. Due to the tree structure of the system, there exists exactly one path between any two nodes. Consequently, a message sent in the system can eventually reach all modules.

In Figure 1, if the keyboard driver wants to read an I/O port, this is accomplished by sending a respective message to the microkernel, which provides an abstraction of the I/O subsystem. The message is sent over the keyboard driver's parent bus and reaches both the *VGA driver* and the *Drivers* gateway. The *VGA driver* does not provide the requested service and ignores the message. The *Drivers* gateway can apply policy checks to ensure that the message will not harm the system if forwarded to a lower system level, i.e. a region of higher criticality and, thus, higher trust. Eventually the message is received and processed by the microkernel, which then sends a reply message.

C. Flexibility and robustness/trust

HM flexibility aspects: Targeting design flexibility, a high degree of modularity due to small and isolated modules enables the reuse of a large fraction of the code base. Upon change of the hardware architecture, only those modules that directly implement hardware abstractions need to be re-implemented. Even in these modules, specific aspects of hardware abstractions can be properly modularized by implementing them as libraries, similar to library OSs.

Contrary to library OSs, HMOSs are also intended to provide high run-time flexibility. A high degree of module isolation is one prerequisite for hot-swapping system functionality. Furthermore, broadcast communication across modules on the same bus simplifies the state transfer of modules by facilitating the monitoring of module interactions and thereby module state inference. Interposition is easily accomplished, as parent modules control where child modules are attached in their sub-hierarchy.

HM robustness and trust aspects: The HM design provides robustness as a direct consequence of the loose coupling between small and isolated components, since this limits the possibility of error propagation from faulty or vulnerable modules to other modules. The hierarchical structure furthermore enforces error containment to system sub-hierarchies, thereby preventing defects in "less operation-critical" components from affecting "more operation-critical" components in lower levels. By organizing system components in a hierarchy rather than a flat structure, it is possible to place components according to their level of trust, where trust comprises both reliability and security aspects. For example, device drivers are usually provided by hardware manufacturers rather than OS developers. This leads to varied degrees of trust in different components of the system. For existing operating systems, drivers either run at the same privilege level as the OS or as untrusted user applications, where a positive impact on trustworthiness is traded for a significant performance overhead. In many modern systems considerable effort is spent on additional sandboxing mechanisms for the inclusion of untrusted third-party components at the same privilege level as a trusted component, both in the kernel (e.g. [13], [14]) and in user processes (e.g. [15]). Our architecture natively supports a *hierarchy of trust*, that allows fine-grained trade-offs between the assurance of dependable operation and the run-time overhead for components.

IV. Summary and Outlook

In this paper we have introduced a novel OS architecture that provides high degrees of design and run-time flexibility to reflect the rapidly changing requirements imposed on future OSs. Flexibility is achieved by a number of fundamental design choices: fine-grained system composition, message-passing, bus-local broadcast communication and hierarchical organization. We argue that these design choices also benefit robustness and trust. We are currently implementing an OS according to the proposed structure on x86 and on customized hardware to assess the performance implications of our design choices and how these are affected by properties of specific hardware architectures.

References

[1] D. A. Holland and M. I. Seltzer, "Multicore OSes: looking forward from 1991, er, 2011," in *Proc. HotOS'11*, 2011, pp. 33–33.

[2] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *Proc. SASP'08*, 2008, pp. 101 –107.

[3] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10 – 16, 2005.

[4] IEEE, "Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, p. 1, 1990.

[5] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, Apr. 2007.

[6] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, "K42: building a complete operating system," in *Proc. EuroSys'06*, 2006, pp. 133–145.

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proc. SOSP'95*, 1995, pp. 251–266.

[8] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. S. Rosenburg, and J. Xenidis, "System Support for Online Reconfiguration," in *Proc. USENIX ATC'03*, 2003, pp. 141–154.

[9] M. Segal and O. Frieder, "On-the-fly program modification: systems for dynamic updating," *Software, IEEE*, vol. 10, no. 2, pp. 53 –65, 1993.

[10] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," in *Proc. OSDI'08*, 2008, pp. 43–57.

[11] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "An operating system for multicore and clouds: mechanisms and implementation," in *Proc. SoCC'10*, 2010, pp. 3–14.

[12] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proc. SOSP'09*, 2009, pp. 29–44.

[13] M. M. Swift, "Improving the reliability of commodity operating systems," Ph.D. dissertation, University of Washington, 2005.

[14] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proc. SOSP'09*, 2009, pp. 45–58.

[15] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proc. SSP'09*, 2009, pp. 79–93.