The Hierarchical Microkernel: A Flexible and Robust OS Architecture

Stefan Winter, Martin Tsarev, Neeraj Suri
DEEDS Group, TU Darmstadt
moduli-os@deeds.informatik.tu-darmstadt.de

1 Introduction

Mediating across user application requirements and hardware capabilities, OSs are required to adapt if either changes. Classical OS architectures usually match a limited spread of hardware configurations and application requirements (i.e., they constrain *design flexibility*) and tolerate limited changes in their execution environments during operation (i.e., they 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.

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" [3], 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. HM systems are composed from small exchangeable components, called *modules*.
2. Inter-module communication is solely based on *message-passing* for loose coupling and a *local broadcast* communication paradigm across neighboring modules enables 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, to confine the overhead of broadcast communication to the respective subsystems.
4. The parent-child relationship in the mono-hierarchy 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.

2 HM Architecture

The HM composition from the two core building blocks – modules and buses – is depicted in Figure 1. *Modules* are self-contained functional entities, resembling both active or passive elements (e.g., processes or libraries) found in 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.

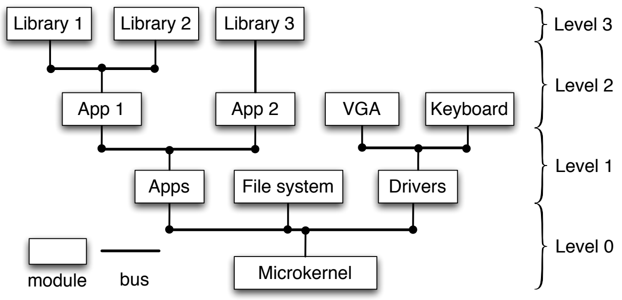

Figure 1: Example of a HMOS

2.1 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 physical bus-based communication networks. 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 and pursue communication. Broadcast communication also facilitates 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 [1, 5]. 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.

2.2 HM's hierarchical organization

To avoid centralized resource management, which could become a performance bottleneck (especially with a broadcast communication paradigm) or a single point of failure, our architecture follows a divide-and-conquer strategy 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.

3 HM Flexibility and robustness/trust aspects

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.

HM OSs 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 undetected error propagation from faulty or vulnerable modules to other modules. The hierarchical structure facilitates error containment to system subhierarchies, thereby preventing defects in "less operation-critical" components in the upper levels from affecting "more operation-critical" components in the 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. [4, 2]) and in user processes (e.g. [6]). Our architecture natively supports a *hierarchy of trust*, that allows fine-grained trade-offs between the assurance of dependable operation and run-time overhead for components: A position in the upper system levels provides higher isolation, but imposes communication latencies because of (a) runtime monitoring of the possibly erroneous/malicious actions of the module and (b) the necessity of message forwarding/routing by gateway modules and buses. Such isolation/performance trade-offs can be adjusted at runtime through module relocation to other system levels.

References

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

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

[3] IEEE. Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990* (1990), 1.

[4] Swift, M. M. *Improving the reliability of commodity operating systems.* PhD thesis, University of Washington, 2005.

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

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