Global Platform Management by Using Synchronous Device Drivers in µ-Kernel-based Systems

Nicolas Berthier
UJF/Verimag [*]
nicolas.berthier@imag.fr

Florence Maraninchi
Grenoble INP/Verimag
florence.maraninchi@imag.fr

Laurent Mounier
UJF/Verimag
laurent.mounier@imag.fr

Abstract

We prospect a solution for the problem of designing µ-kernel-based embedded systems. This approach relies on a global platform management mechanism made of synchronous device drivers.

1. Context

One can identify two major classes of operating system kernels currently in use in embedded systems: monolithic, and µ-kernels. Yet µ-kernels gain more and more attention in the field of embedded systems, due to numerous design advantages. However, power management, security or even safety properties, are critical aspects for the construction of embedded systems. We propose a solution making the enforcement of global resource management for µ-kernel-based embedded systems easier.

1.1 Monolithic Kernels and Major Related Problems

In monolithic kernel systems, components such as scheduler, device drivers and system services (network stack, virtual file systems) form a single binary blob that runs in *kernel space*: each of its parts executes with (almost) all the privileges required to access and control the full hardware platform. As a result, no protection barriers enforce isolation between these components. Major problems that occur are the following:

- A buggy kernel component (*e.g.,* device drivers and system services), or even worse, an unanticipated behavior of the composition of several cooperating ones, can cause a crash of the complete system. Such errors can also cause security issues, since being able to execute cleverly written malicious code in one single part of this kernel allows to take over the entire machine.
- Writing this kind of huge programs is also very error-prone: Ostrand et *al.* [11] determined that in large-scale software projects, $1,000$ lines of code usually contain from 1 to 16 bugs. Although these figures are a rough estimate, we can deduce that monolithic kernels, commonly made of millions of lines of code, are likely to contain many bugs.

Examples of widespread monolithic kernels comprise Linux, OpenBSD or Windows' kernel.

1.2 µ-Kernels

µ-Kernel-based systems try to address some of the problems faced by monolithic kernels by leveraging hardware isolation mechanisms. They usually involve a µ-kernel that manages a small set of abstractions, and possibly several unprivileged processes (often called *servers*) that run in *user land* (*i.e.,* not in kernel space). Application processes use services provided by the µ-kernel and the servers. Classical hardware mechanisms are used so as to enforce isolation between all these processes; an operating system based on a monolithic kernel would do it for application processes only.

Intrinsically, µ-kernels provide mechanisms for a well-structured user land, that constitutes a considerable advantage for the design of complex systems. A well-suited design of such system necessarily makes a clear separation between the roles of each server, and involves the definition of neat specifications of inter-process communication protocols. Then, each server can be implemented separately, thus simplifying its design *a priori*.

1.3 A Short History of µ-Kernels: The Dawn of L4

The first generation of µ-kernels, whose main representative is Mach [1], revealed to be relatively slow compared to monolithic counterparts.

Later, Liedtke [9] designed the L4 specification, based on the idea that the kernel should provide only a minimal set of abstractions; any unnecessary addition would then reside in user land. He also developed a highly optimized, though non-portable, implementation of this specification: performance results turned out to be comparable with those of its contemporary monolithic kernels.

More recently, Härtig et *al.* [7] and Heiser [6] further investigated the concept of *trusted computing base*, and put forward the usage of L4 as a virtual machine monitor for, *e.g.,* running a paravirtualized Linux instance with legacy applications, and other L4 tasks side by side.

Klein et *al.* [8] developed seL4, a formally verified implementation of L4, including extensions such as a capability model for *resource access control*. Ruocco [12] studied the potential usage of L4 in the context of real-time applications. He concluded that L4 presents numerous advantages when designing real-time applications, along with appropriate device drivers that run in user land.

2. Problems

2.1 Device Drivers Integration

Studying Linux and OpenBSD, Chou et *al.* [5] stated that device drivers significantly contribute to the number of bugs in operating system kernels. Thus, one problem that still arises when using L4 as an operating system basis, is the integration of *device drivers* in the whole system.

In their experiments about the trusted computing base, Härtig et *al.* [7] proposed to leverage already written device drivers from the paravirtualized Linux so as to drive actual peripheral devices, at the expense of losing the possibility of sharing such device among several L4 tasks or instances of Linux. Another solution they propose for reusing Linux device drivers is to integrate them into

[*] Verimag is an academic research laboratory affiliated with: the University Joseph Fourier Grenoble (UJF), the National Center for Scientific Research (CNRS) and Grenoble Polytechnic Institute (Grenoble INP).

a glue, called *Device Driver Environment*, that produces a driver running in user land and using L4 services to access the hardware.

The previous approach is similar to the one of Apple. Their XNU kernel is built on top of a first-generation µ-kernel (from the Mach family). However, it involves additional pieces of code running in kernel space: one provides UNIX abstractions to user-level processes; another is the *I/O Kit* framework, that includes device drivers. The latter supports dynamic loading and unloading of drivers into the kernel, along with possible interactions with user land device drivers. At the end of the day however, such a kernel looks more like a monolithic one.

2.2 Global Knowledge Problem

Another problem that arises (but is not specific to) µ-kernel-based systems, is the lack of *global knowledge* about the state of the whole system.

The "suspend blockers" case: The controversial *suspend blockers* (*aka* wakelocks)[1] case is a typical example of the need for global knowledge, that occurs in the context of Linux-based embedded systems. One of its parts that is related to device drivers could be formulated as follows. When the kernel notices there is no computation to be performed, it can decide to put CPUs asleep. Current practice is to only put the CPUs into so called *idle* mode. However, Android developers have proposed to "opportunistically" use deeper sleep modes that globally affect the platform as a whole (*e.g.,* ACPI full system suspend, that also shuts down some peripheral devices) in order to save more power when there is neither computation to be performed, nor device in use. Yet, this feature supposes that the kernel can decide if there is no currently working device that should not be suspended. Suspend blockers are structures proposed to circumvent this kind of problems. They should be used by device drivers to provide this information to the Linux kernel.

The key idea behind this case is that one needs some *global knowledge* about the state of each peripheral device so as to decide whether one can suspend the whole platform. This knowledge is actually distributed among the device drivers, since each one is in principle the only manager of a part of the platform.

3. Approach Proposal

We have already published some ideas concerning synchronous programming of device drivers in the context of wireless sensor network nodes [3]. We advanced the usage of a *Control Layer* (CL), comprising synchronous device drivers automata, whose parallel composition constitutes a knowledge about the state of the whole system. Those automata can be programed using a synchronous language [2], then compiled into sequential code. Further investigating the extensions of our approach led us think about its usage in the context of µ-kernel-based operating systems.

We propose to integrate all device drivers in a single user land server, following a similar interaction method that was used for the CL (that could become a *Platform Control Server* — PCS). This server could *react* to requests from other tasks and the hardware, and then maintain a global knowledge of the full system. Considering the "suspend blockers" case, it could then be relatively easy to decide whether it is possible to suspend the full system (thanks to the synchronous view of the underlying global state).

Advantages: This approach shares common points with the *I/O Kit* framework used by Apple, and the *Device Driver Environment* proposed by Härtig et *al.* [7], in the sense that it takes the best of both worlds: it combines the advantages of user land device drivers with the global knowledge that can be gathered by grouping them.

In addition, global control could be integrated and handled properly, as we did for WSN applications: in [3], we proposed to introduce a *controller* enforcing global properties, such as forbidding several peripheral devices to consume a high amount of power at the same time. In the context of embedded systems we consider presently, one could also design booking controllers (*i.e.,* that queue requests for deferred processing) since memory and computation time are less constrained resources than in WSN nodes.

Challenges: One challenging aspect resides in the potential dynamism: how could we manage device hotplug? We claim that preserving the synchronous aspect of the PCS is a key point, since this helps keeping a coherent knowledge of the global state. Investigating *reactive programming* [4] of device drivers could help on this point. A related problem concerns the reuse of device drivers already written for other systems. This could reveal feasible by using an encapsulation of such code into some glue, and extracting the corresponding device driver automaton. Also, synthesis of drivers from appropriate specifications is conceivable, as Ryzhyk et *al.* [13] did for monolithic kernels.

Eventually, this new approach for synchronous device drivers integration and design in µ-kernel-based embedded systems presents several advantages, by keeping track of the state of the full platform. We believe that such a proposition could lead to interesting perspectives in the design of embedded systems.

References

[1] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Summer*, pages 93–113, 1986.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[3] N. Berthier, F. Maraninchi, and L. Mounier. Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems. In *Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, Chicago, IL, USA, 2011.

[4] F. Boussinot and R. de Simone. The SL Synchronous Language. *IEEE Trans. Software Eng.*, 22(4):256–266, 1996.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An Empirical Study of Operating System Errors. In *SOSP*, pages 73–88, 2001.

[6] G. Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM.

[7] H. Härtig, M. Roitzsch, A. Lackorzynski, B. Döbel, and A. Böttcher. L4- Virtualization and Beyond. *Korean Information Science Society Review*, 2008.

[8] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In Matthews and Anderson [10], pages 207–220.

[9] J. Liedtke. On µ-Kernel Construction. In *SOSP*, pages 237–250, 1995.

[10] J. N. Matthews and T. E. Anderson, editors. *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009.

[11] T. J. Ostrand and E. J. Weyuker. The distirubtion of faults in a large industrial software system. In *ISSTA*, pages 55–64, 2002.

[12] S. Ruocco. A Real-Time Programmer's Tour of General-Purpose L4 Microkernels. *EURASIP J. Emb. Sys.*, 2008, 2008.

[13] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with termite. In Matthews and Anderson [10], pages 73–86.

[1] Please refer to the "Suspend block" article by Jonathan Corbet (`http://lwn.net/Articles/385103/`), and the `linux-pm` mailing list for further information about this topic.