Trusted Collaborative Real Time Scheduling in a Smart Card Exokernel

Damien Deville, Christophe Rippert, Gilles Grimaud

▶ **To cite this version:**

Damien Deville, Christophe Rippert, Gilles Grimaud. Trusted Collaborative Real Time Scheduling in a Smart Card Exokernel. [Research Report] RR-5161, INRIA. 2004. inria-00077048

HAL Id: inria-00077048
https://inria.hal.science/inria-00077048

Submitted on 29 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Trusted Collaborative Real Time Scheduling in a Smart Card Exokernel

Damien Deville — Christophe Rippert — Gilles Grimaud

N° 5161

Avril 2004

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5161--FR+ENG

Trusted Collaborative Real Time Scheduling in a Smart Card Exokernel

Damien Deville * , Christophe Rippert * , Gilles Grimaud *

Thème 1 — Réseaux et systèmes
Projet POPS

Rapport de recherche n° 5161 — Avril 2004 — 15 pages

Abstract: This paper presents the work we have conducted concerning real time scheduling in Camille, an exokernel dedicated to smart cards. We show that it is possible to embedded a flexible real-time operating system despite the important hardware limitations of the smart card platform. We present the major difficulties one has to face when integrating real time support in an exokernel embedded on a very resource-limited platform. We first present a naive solution consisting in allocating an equal time slice to every system extensions and letting each one share it as needed amongst its tasks. We show that this solution does not account for loading of new extensions in the system, and that it can fail if some extensions have much more work to carry out than the others. We then present a more complex solution based upon collaborative schedulers grouped as virtual extensions. We show that this solution supports dynamic loading of new extensions and works even for very unbalanced task repartitions. We finally address the issue of trust between the collaborating extensions and we propose a solution based on exhaustive testing and formal proving of the plan functions.

Key-words: Real time, collaborative scheduling, smart card, exokernel

* IRCICA/LIFL, Univ. Lille 1, UMR CNRS 8022, INRIA Futurs, POPS research group. This work is partially supported by grants from the CPER Nord-Pas-de-Calais TACT LOMC C21, the French Ministry of Education and Research (ACI Sécurité Informatique SPOPS), and Gemplus Research Labs.

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 —Télécopie : +33 1 72 92 59 ??

Ordonnancement temps-réel collaboratif dans un exonoyau pour cartes à puce

Résumé : Ce document présente le travail que nous avons réalisé concernant l'ordonnancement temps-réel dans Camille, un exonoyau dédié aux cartes à puce. Nous montrons qu'il est possible d'embarquer un système d'exploitation temps-réel et extensible dans des dispositifs matériels aussi contraints que les cartes à puces. Nous présentons les difficultés principales rencontrées lors de l'intégration du support temps-réel dans un exonoyau embarqués dans des dispositifs très contraints. Nous détaillons tout d'abord une solution naïve consistant à allouer un quantum de temps à chaque extension système et laisser chacune d'elle le répartir entre ses tâches. Nous montrons que cette solution ne supporte pas l'insertion dynamique d'extensions dans le système, et qu'elle peut échouer pour les extensions ayant beaucoup plus de travail à réaliser que les autres. Nous présentons ensuite une solution plus complexe basée sur la collaboration d'ordonnanceurs regroupés dans des extensions virtuelles. Nous montrons que cette solution supporte le chargement dynamique d'extensions et fonctionne même pour des répartitions de tâches très déséquilibrées. Nous traitons finalement le problème de la confiance entre les extensions collaborantes, et proposons une solution basée sur le test exhaustif et la validation formelle des fonctions de plan.

Mots-clés : Temps réel, ordonnancement collaboratif, cartes à puce, exonoyau

1 Introduction

Smart card operating systems have to face very hard constraints in terms of available memory space and computing power. Thus, most on-card operating systems are usually based on ad hoc architectures devised to fit completely to the underlying hardware, at the price of portability of the system. We believe that the exokernel architecture is an interesting alternative for resource-limited platforms such as smart cards. It enables the system programmer to choose which system abstraction he wants to include in the kernel, and thus permits him to build a fully-customized operating system including only needed services. This is especially important for platforms as smart cards which are so limited by hardware constraints that no resource waste can be tolerated.

However, the exokernel as presented by Engler in [1] does not include any support for real time scheduling. This is a major drawback for an operating system dedicated to smart cards since the specifications of most smart card platforms impose strict deadlines for communications between the card and the terminal to which it is connected. This advocates the real time paradigm to guarantee response times.

This work overlaps several domains which are usually not combined, since it is concerned with extensible kernel architectures, embedded systems, dynamic loading of scheduling policies and real time systems. Some work has already been conducted on the use of exokernel-like architectures for embedded devices [2, 3], however this work usually does not address real time issues. The Bossa [4] platform permits to dynamically load new schedulers in a running system, but is not dedicated to embedded systems. Similarly, the research done concerning WCET computation in the RTEMS real time operating system [5] can be useful in our context but it has been conducted on a monolithic kernel with very different architectural issues than those of exokernels.

We first present the context of this work by discussing the major constraints of smart cards and the advantages of using the exokernel architecture on resource-limited devices. We then present the Camille exokernel and the extensions we have implemented to support real time scheduling. We propose a simple solution to find a scheduling plan for the various tasks in the system and exhibit its shortcomings concerning dynamic loading of new extensions and its unsuitability for systems where some extensions have much more work to carry out than others. We then present a more elaborate solution based on collaborative schedulers and show that it preserves the extensibility of the system and succeeds in finding a scheduling plan even for systems where the task repartition is very unbalanced. We finally discuss the issue of trust between the collaborating extensions and prove that it can be solved by combining exhaustive testing and formal proving of the scheduling plan.

2 Working context

2.1 Smart cards

Smart cards are small, portable and secured devices characterized by the very strict hardware constraints they impose to operating system programmers. Most of these constraints come from the necessity to make smart cards tamperproof [6] as defined by the ISO standard [7].

Microprocessors used in smart cards range from old 8-bit CISC microchip (4.44 Mhz) to powerful 32-bit RISC processor (100 to 200 Mhz). The type of CPU which can be used in a smart card is strongly influenced by the ISO [7] constraints which impose a maximum thickness to guarantee tamper resistance. Historically, smart card manufacturers used 8-bit processors because they induced more compact operating systems and applications code. But smart cards now need to be more efficient and new embedded applications require more and more computing power. Thus card designers sometimes choose 32-bit RISC processors (or enhanced 8/16 bits CISC CPU). The computing power of these processors is sufficient for most treatments needed by smart card applications. More complex algorithms like for cryptographic functions typically use dedicated coprocessors. The simplicity of smart card processors (*e.g.* no data/instruction cache, no pipeline, etc) greatly simplifies the computation of applications Worst Case Execution Time (WCET) which is needed for real time scheduling.

Different types of memory coexist in a smart card. Random Access Memory is used as a working space by the system and the applications. Read Only Memory typically contains the kernel of the operating system which usually does not change after having been loaded on the card. Finally, some kind of persistent rewritable memory (typically EEPROM or FlashRAM) serves as a storage space for data which must be preserved when the card is disconnected. Due to the maximum thickness of a card specified by the ISO standard, the total amount of memory available is usually very small. A modern smart card typically disposes of 2–4 KB of RAM, 32–128 KB of persistent rewritable memory and 64–128KB of ROM. Due to the very limited amount of RAM available, the persistent rewritable memory is sometime used as a working space. This poses a problem for a real time system since the access time of RAM and EEPROM for instance are very different (EEPROM being mush slower that RAM). Moreover, each write in persistent memory requires a hardware lock of the corresponding memory zone which halts the processor until the end of the write. Thus the system must know in which kind of memory an accessed value is stored to take into account the delay induced by the memory access in the WCET computation.

I/O protocols are also defined by ISO standards. ISO 7816 defines protocols for wired smart cards (*i.e.* smart cards that need to be physically connected to a terminal to communicate) whereas ISO SC17 14443 specifies protocols for contactless smart cards. Typical communication protocols provide 9600 to 192000 bauds transfer rates over a half-duplex serial line. I/O protocols are responsible for many real time constraints. For instance, ISO 7816-3 and ISO 7816-4 define precisely the maximum time for the card to send an Anwer To Reset packet when it is switched on, to notify the terminal that it is indeed connected and working properly. Similar deadlines must be enforced when the terminal sends a request to

the card which must respond in less than a fixed time, otherwise the terminal may consider that the card has been disconnected and switch its power off. This advocates the use of a real time operating system which permits to guarantee the execution time of processes in charge of handling requests. Another important requirement is a way to demultiplex hardware interrupts in the operating system since most communications are usually done using interrupts. This is very troublesome for a real time system since a hardware interrupt suspending the execution of a process can thoroughly disrupt a scheduling plan.

2.2 Camille

Camille is an extensible operating system designed for resource-limited devices, such as smart cards for instance. It is based on the exokernel architecture [1, 8] and advocates the same principle of not imposing any abstractions in the kernel, which is only in charge of demultiplexing resources. Camille provides secure access to the various hardware and software resources manipulated by the system (*e.g.* the processor, memory pages, native code blocks, etc) and enables applications to directly manage those resources in a flexible way.

The exokernel architecture is well-suited for resource-limited devices such as smart cards. Starting from a minimal kernel and adding only needed system abstractions permits to build small systems including only the services that the applications will actually use. Moreover, the flexibility of the exokernel architecture permits to dynamically add new applications or services that were omitted when the system was built, thus supporting post-issuance of applications and kernel extensions. Finally, it permits to fully-customize the implemented services for the platform on which they will run and for the targeted applications, since the programmer developing them is the one who will use them to build the system. Thus, abstractions can be programmed to fit at best the needs of the applications and fully exploit the resource available.

Figure 1 describes the split architecture of Camille.

System components and applications can be written in a variety of languages (including Java, C, etc). The source code is translated in a dedicated intermediate language called Façade [9] by appropriated tools (*e.g.* a Java to Façade converter or the Gnu Compiler Collection for C code). Using an intermediate language enhances the portability of the various components is a way similar to Java bytecode. This portability is strengthened by the very limited number of machine-dependent system primitives, which can easily be ported from one platform to another and are grouped in a component called the Base system. To guarantee the efficiency of the system and the applications, the Façade code is translated into native code using an embedded compiler using just-in-time compilation techniques (referenced as the native code generator component in Figure 1). This compiler converts Façade programs when they are loaded in the card, and performs machine-dependent optimizations to exploit fully the underlying hardware. Some machine-independent optimizations are also performed when the code is translated from the source language to Façade to benefits from the computing power of the workstation on which the system is built, which is obviously far greater than the computing power of a smart card.

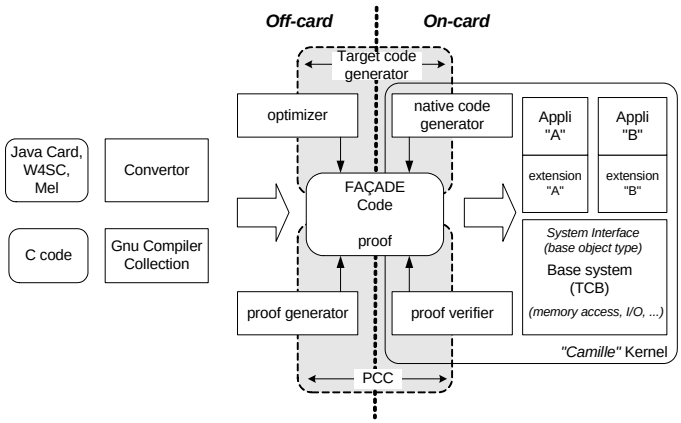

Figure 1: Camille architecture and software infrastructure.

FAÇADE is a very simple language which can easily be type-checked. To guarantee the integrity of the embedded code, the proof of the type-correctness of each program is computed by a proof-generator and included in it when it is loaded on the card. This Proof-Carrying Code [10] is verified on-card before being translated into native code [11]. This ensures that no program loaded on the card will compromise the integrity of the system. This is essential for an operating system dedicated to smart cards, which have very strong security requirements.

Thus, Camille benefits from the following properties:

Portability thanks to the use of the intermediate language FAÇADE

Security since all programs loaded in the card are type-checked before being translated to native code

Extensibility due to its exokernel architecture, which imposes no system abstractions in the kernel and permits to add new services as needed by embedded applications

A more detailed presentation of Camille can be found in [12], including benchmarks and experimental results. The Camille prototype demonstrates the feasibility of an extensible operating system dedicated to smart cards considering its moderate memory footprint (17 KB of native code including 3.5 KB for code verification, 8.5 KB for native code generation, and 5 KB for hardware multiplexing). The remainder of this paper presents our work on real time issues in the context of the Camille exokernel. We propose an architecture to support real time schedulers in an exokernel and also some clues to enable schedulers to share their CPU quantum.

2.3 Real time extensions for Camille

The standard exokernel architecture, as defined by Engler in [1], does not offer any dedicated real time primitives since it is only concerned with supporting extensibility while ensuring maximum security. The scheduler included in the kernel only guarantees equity of access to the processor to each subsystem (or *extension* in the exokernel terminology). This scheduler uses a round-robin policy to elect an application (and provides a *yield* primitive for extensions that want to relinquish the remaining time of their slot to others). The main motivation of a real time version of Camille (that we call Camille RT) is therefore to permit applications and system extensions to implement real time primitives, and to make standard applications coexist with real time ones.

Supporting real time issues in an extensible embedded operating system implies to be able to:

(*i*) quantify the execution time for each of the exokernel primitives (*e.g.*, delay related to a virtual TLB access, locks during EEPROM writing, etc),

(*ii*) find a schedule that satisfies all running applications deadlines,

(*iii*) define a way to allow applications to notify their real time requirements (deadline, rate, start time, ...) to the exokernel,

(*iv*) quantify the execution time for each real time block of code expressed in the FAÇADE intermediate language.

The first three points concern the real time problematic in a standard exokernel while the last one is specific to Camille which uses the FAÇADE intermediate language to increase portability. The first and last points are issues known in the real time community as *Worst Case Execution Time* computation. WCET analysis is known to be a difficult problem on a smart card, since WCET computation must usually be performed off-card by complex algorithms requiring large amounts of memory. Furthermore, the use of the FAÇADE intermediate language in Camille complexifies WCET computation. The embedded on-the-fly compiler translates FAÇADE code into native one, which can invalidate results found on FAÇADE programs. Thus WCET computation needs to be distributed between off-card tools and the on-card compiler. Since smart cards microprocessors are limited as explained in Section 2, compilation patterns used by embedded compilers are usually simple enough to clean up WCET computation. But the main problem lies in predicting write delays in EEPROM, which induce locking of the processor as mentionned before. These writes can be bound to 2 ms per page, even if this simplification does not take into account tasks interleaving. Thus, precise WCET computation on a smart card is an open problem which goes beyond the scope of this article and that we shall consider as solved in the rest of this paper.

3 Proposal

We present in this section the two approaches we have tested to support real time scheduling in Camille. The first scheme is a naive solution based on the original exokernel architecture. We show its limitations and then propose a more complex solution based on collaborative schedulers.

3.1 Blind scheduling

A simple solution for supporting real time scheduling in a exokernel is to follow the microprocessor demultiplexing approach presented by Engler in [1]. This scheme, that we shall call blind scheduling thereafter, provides equity in terms of access to the CPU by using a simple and impartial round-robin policy. Each extension is granted access to the micro-processor with a rate proportional to the number of running extensions. An extension provided with a simple round-robin policy can support non-real time applications within its time slot. Let N represent the number of extensions, each extension has the guarantee to have access to the microprocessor with a rate of $\frac{1}{N}$. This guarantee allows an extension to support real time tasks within its time slot. Once the extension is granted a time slot, the chosen extension can use its scheduler to elect one task to be run. Figure 2 gives an example of an exo-scheduler supporting two extensions (one which is in charge of scheduling tasks A and B, and the other one supporting the task C).

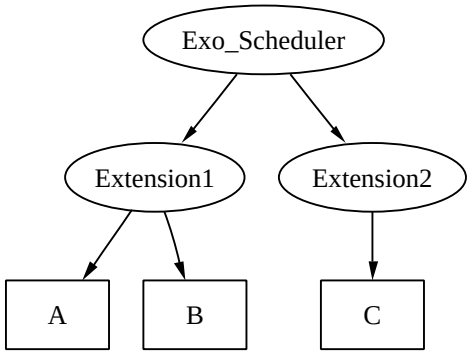

Figure 2: A simple hierarchical scheduling architecture.

This two-level CPU sharing process is close to the hierarchical scheduler approach [13] or to the solution described in [14] to support a mix of real time and general purpose tasks. It is sufficient to build a real time scheduler over an exokernel, but it also introduces sub-optimal solutions. Hardware interrupt demultiplexing also causes a problem with this solution since interrupt notifications will be delayed until all real time tasks are completed

by the destination extension. In the worst case an extension may receive the interrupt notification after $N-1$ time slices. An immediate solution is to reduce the size of the quantum, which increases the reactivity of the system toward hardware interrupts but also induces more context switches and might compromise the performances of the system.

Another difficulty arises when a new extension is loaded: the $\frac{1}{N}$ rate guarantee must be reconsidered since the number of extensions has increased and the rate should be changed to $\frac{1}{N+1}$. A vote must then be cast to determine if every extensions accept the new rate. If the vote succeeds, the new extension is accepted and added to the extensions list right after the end of the current round-robin round. Otherwise, the user is notified of the failure.

Figure 3 illustrates the main problem of blind scheduling. Considering two extensions E1 and E2, each one is granted access to the processor alternatively since we use a round-robin policy for the root scheduler. E1 manages two tasks, A and B, with an execution time of one quantum for each one, and with deadline of respectively two and five (*i.e.* task A requires one quantum every two quantum, and task B needs one quantum every five quanta). E2 manages only one task executing in one quantum and with a deadline of four. This system is thus periodic with a period of twenty quanta.

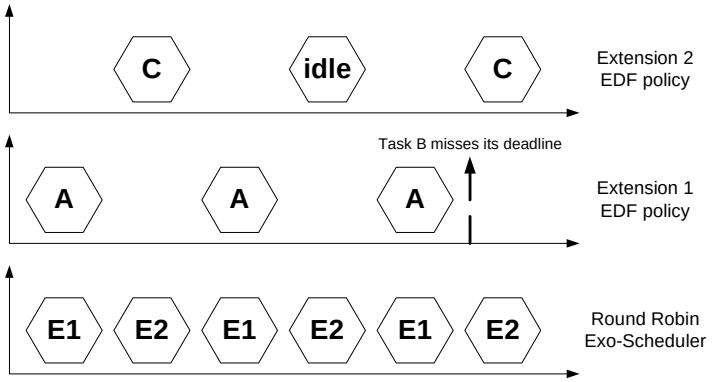

Figure 3: Failure of the blind scheduling.

The round-robin exo-scheduler alternatively gives one quantum to E1 and E2 which both use an Earliest Deadline First [15] scheduling policy. According to this policy, E1 must give the fifth quantum to task A since it has an earliest deadline than B, which causes task B to miss its deadline. This illustrates the shortcomings of blind scheduling since E2 had an idle time slice which B could have used. With this solution, extensions only take into account their own tasks and do not yield unused time slices to other extensions. In the example presented in Figure 3, a simple Rate Monotonic policy [15] would have succeeded in scheduling tasks A, B and C if collaboration between extensions could have been assured (the successful scheduling plan would have been ACAB-ACAB-ACAB-ACAB-ACA*Free*).

3.2 Collaboratives Schedulers

The previous solution fails because the Camille extension supporting task A and B is isolated from the one supporting task C. Another approach consists in grouping real time extensions that accept to share their time slice into a virtual collaborative exo-scheduler. This exo-scheduler guarantees that the deadlines of each tasks under its supervision will be met even if extensions do not trust each other. This collaborative architecture is presented in Figure 4.

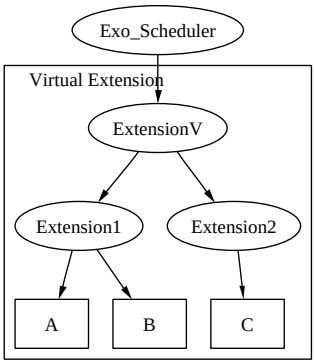

Figure 4: A collaborative scheduling architecture.

Extensions that want to share their access to the processor provide a plan function representing its scheduling policy. This function takes as parameters a set ot tasks, the hyperperiod of the system (*i.e.* the least common multiple of the periods of all the tasks, since a set of periodic tasks can be seen as a periodic system of period equal to the hyperperiod of the system) and the current time. It is in charge of electing the task which will run for the next time slice. One scheduler of a collaborative extension is granted the right to schedule all tasks of all collaborative extensions and is named the active scheduler. When the virtual extension is given a time slice, it asks the active scheduler to select the task to be executed for this time slice. The virtual extension then requests the owner of the chosen task to activate it. A notification is sent to the owner of the chosen task when the end of the time slice is reached. Figure 5 illustrates a successful scheduling plan for the same example presented in Figure 3, but this time using collaborative scheduling.

It is important to be able to guarantee to extensions that their deadlines will be met by the active scheduler, which can be part of another extension whom they do not always trust. This is achieved thanks to a checking algorithm used during the loading and installation phases.

When loading a new collaborative real time extension, a new vote is called. If the vote succeeds, the collaborative exo-scheduler verifies that the new extension can be installed along with its scheduler and tasks. The main difficulty is to find in the pool of collaborative

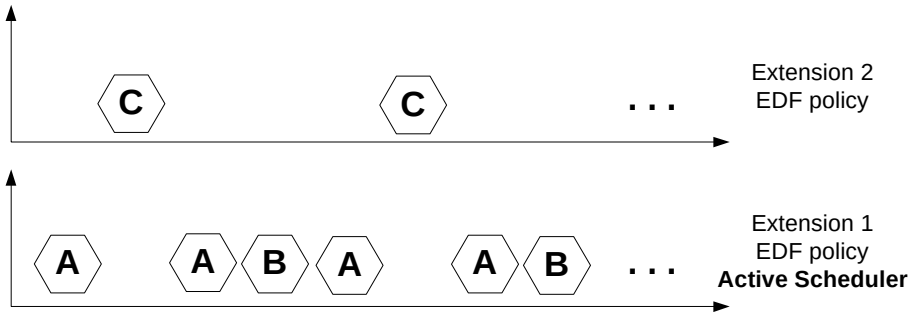

Figure 5: A successful scheduling plan with collaborative scheduling.

schedulers a new scheduler which will meet the deadlines for the new set of tasks and to ensure that this scheduler can be trusted. The plan functions are used to find a suitable scheduler. Each potential scheduler is submitted a tasks list containing all the current tasks and the ones coming with the new extension. If it is not able to schedule this task set, the collaborative exo-scheduler asks the next potential scheduler until one manages to find a successful scheduling plan or the last one fails (which means that the new set of tasks is impossible to schedule according to the extension scheduling policy). Once the collaborative exo-scheduler has found a suitable scheduler, it verifies that it can be trusted by asking it to find a scheduling plan on the whole system period (*i.e.* the hyperperiod). This verification is of complexity $O(hyperperiod)$ and consists in checking that every task is granted a time slice big enough to be able to finish before its deadline. The schedule plan obtained through this verification is not stored in memory as it would use to much of that limited resource.

3.3 Trust issues

An important security issue must also be considered. With blind scheduling, extensions are isolated from each others which implies that they are protected from a malicious extension trying to disrupt the schedule plan. This is not the case with collaborative scheduling, since they need to share their access to the processor to collaborate. This raises a question of trust between the collaborating extensions. Plan functions can be used to address this issue. They are useful during two different phases, as illustrated in Figure 6.

First, each plan function is exhaustively tested on the hyperperiod of the task set to schedule, when the extension which exports it is loaded in the system. As soon as a plan function succeeds this test, the exokernel installs it and starts using it during the execution phase. Installation of the new plan function occurs right after the end of the current round to ensure no task will miss its deadline. If a malicious task is able to detect in which phase it is called, it could cheat by not respecting its schedule plan. Thus, it is necessary to ensure that the plan function is an Untrusted Deterministic Function. An UDF is a

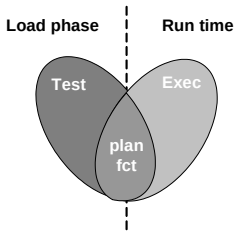

Figure 6: Shared use of the plan functions.

function which is guaranteed to return the same result each time it is called with the same parameters. This implies that it must not keep any internal state and must not access the hardware clock or use any random functions, since this could help it detect when the collaborative exo-scheduler asks for the installation of a new extension or for the election of a task, and could lead to break the guarantee that each real time task will succeed its deadline. A similar problem was solved using a native code analysis in exokernels for the *own()* function which secure the access to metadata for disks management (in Chapter 4 of [1]). Such a property can be checked by using a Domain Specific Language, like for example in the Bossa [4] framework for the development of real time schedulers. In Camille RT, we have modified our code loading process to support UDF verification. An off-card part is in charge of generating a proof that will help the on-card loader verify the UDF property. The integration of the different components to support trusted collaborative scheduling in the Camille RT exokernel is presented in Figure 7.

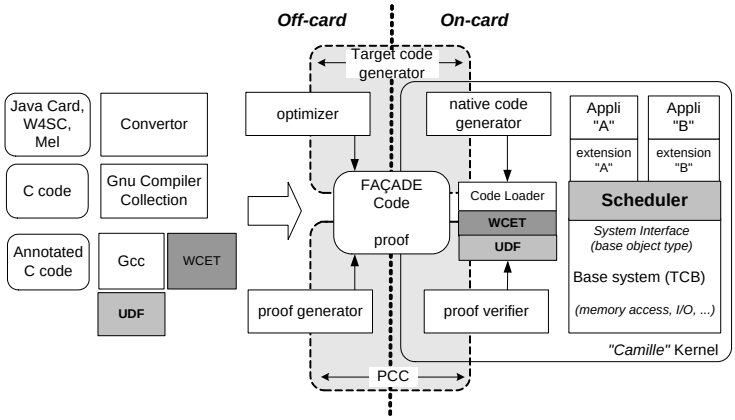

Figure 7: Camille RT architecture supporting trusted collaborative schedulers.

4 Conclusion and future work

We have shown in this paper that it is possible to support real time scheduling in an exokernel dedicated to resource-limited devices as smart cards. We have presented the Camille exokernel, an operating system dedicated to very constrained platforms and shown how it could be extended to support real time scheduling. We have shown that the processor demultiplexing approach proposed in the original exokernel architecture cannot be applied directly to real time scheduling as it can fail for unbalanced task repartitions. We have proposed a solution based on collaborative schedulers that guarantees that deadlines of tasks managed by different extensions will be met and shown how exhaustive testing and formal proving of the plan functions can ensure trust between collaborating extensions.

Work remains to be done concerning the impact of hardware locks on the schedule plan, as they can occur when data are accessed in persistent memory. The problem of efficient interruption demultiplexing and dispatching to extensions must also be addressed as a hardware interrupt raised during the execution of a real time task can disrupt the selected schedule plan. Finally, the issue of WCET computation for very specific platforms as smart cards should be addressed as it remains an open problem.

References

[1] D. R. Engler. *The Exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology (MIT), 1999.

[2] Christophe Rippert and Jean-Bernard Stefani. Building secure embedded kernels with the think architecture. In *Proceedings of the workshop on Engineering Context-aware Object-Oriented Systems and Environments*, November 2002.

[3] Aline Senart, Olivier Charra, and Jean-Bernard Stefani. Developing dynamically reconfigurable operating system kernels with the think component architecture. In *Proceedings of the workshop on Engineering Context-aware Object-Oriented Systems and Environments*, November 2002.

[4] J. L. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in an event type system: the Bossa experience. In *Tenth ACM SIGOPS European Workshop 2002 (EW2002)*, pages 54–61, St. Emilion, France, September 2002.

[5] A. Colin and I. Puaut. Worst-case execution time analysis of the RTEMS real-time operating system. In *13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, The Netherlands, june 2001.

[6] Oliver Kömmerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. In *USENIX Workshop on Smartcard Technology*, pages 9–20, 1999.

[7] International Standard Organisation: ISO. Integrated circuit(s) cards with contacts, parts 1 to 9, 1987-1998.

[8] D. R. Engler and M. F. Kaashoek. Exterminate All Operating System Abstractions. In *the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, USA, 1995.

[9] G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. FAÇADE: A Typed Intermediate Language Dedicated to Smart Cards. In *Software Engineering–ESEC/FSE*, 1999.

[10] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[11] A. Requet, L. Casset, and G. Grimaud. Application of the B formal method to the proof of a type verification algorithm. *HASE*, 2000.

[12] D. Deville, A. Galland, G. Grimaud, and S. Jean. Smart Card operating systems: Past, Present and Future. In *the 5th NORDU/USENIX Conference*, Västerås, Sweden, February 2003.

[13] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[14] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Procedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pages 308–319, December 1997.

[15] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *ACM*, 20(1):46–61, January 1973.

Contents

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399