

Design Rationale of the PURE Object-Oriented Embedded Operating System

*Ute Haack, Wolfgang Schröder-Preikschat,
Friedrich Schön[†], Olaf Spinczyk*

University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
`{olaf,ute,wosch}@cs.uni-magdeburg.de`

[†]GMD FIRST
Rudower Chaussee 5
D-12489 Berlin, Germany
`fs@first.gmd.de`

Abstract

The PURE project aims at providing a portable, universal runtime executive for deeply embedded parallel/distributed systems. The phrase “deeply embedded” refers to systems forced to operate under extreme resource constraints in terms of memory, CPU, and power consumption. The notion “parallel/distributed” relates to the fact that embedded systems are becoming more and more complex in terms of architectural viewpoints. Typical examples are automotive systems. Today’s limousines can be considered as (large scale) distributed systems on wheels. There are cars in daily operation consisting of over 60 networked processors (i.e. μ -controllers) attached to 1–2 MB of global memory. These “decentralized computer architectures” make high demands on both the application and the system software. Dedicated design and implementation principles have to be applied in order to ensure manageability, adaptability, portability, and yet efficiency of the software. In addition, the resulting software structure must promote, and neither hinder nor prevent, the application of formal methods to gain clear statements on correctness and safety of the resulting system complex. This paper discusses design issues of a family-based, object-oriented operating system currently under development and targeting the arena of deeply embedded systems in the above-mentioned sense.

1 Design Approach

Deeply embedded operating systems must exhibit a featherweight system structure that is adaptable to the individual needs of both an application program and

the hardware architecture. The approach followed by PURE is to understand an operating system as a *program family* [3] and to use *object orientation* [5] as the fundamental implementation discipline. The former concept (program families) helps prevent the design of a monolithic system organization, while object orientation enables the efficient implementation of a highly modular system structure.

The program family concept does not dictate any particular implementation technique. A so called “minimal subset of system functions” defines a platform of fundamental abstractions serving to implement “minimal system extensions”. These extensions, then, are made on the basis of an *incremental system design* [2], with each new level being a new minimal basis (i.e., *abstract machine*) for additional higher-level system extensions. A true application-oriented system evolves, since extensions are only made on demand, namely, when needed to implement a specific system feature that supports a specific application. Design decisions are postponed as long as possible. In this process, system construction takes place bottom-up but is controlled in a top-down (application-driven) fashion.

In its last consequence, applications become the final system extensions. The traditional boundary between application and operating system disappears. The operating system extends into the application, and vice versa. Following the pattern of PEACE [4], inheritance is the appropriate technique to either introduce new system extensions or replace existing ones by alternate implementations. Either case, the system extensions are customized with respect to specific user demands and will be present at runtime only in coexistence with the corresponding application. Thus, applications are not forced to pay for (operating system) resources that will never be used.

2 Resource Consumption

The PURE nucleus consists of four building blocks: (1) the *flange*, i.e. abstractions for the attachment of objects to trap/interrupt vectors and propagating exceptional events to higher levels, (2) the *sluice*, i.e. abstractions for the synchronization of interrupt-driven activities in an interrupt-transparent manner (i.e. without both disabling of hardware interrupts and exploiting any special/atomic CPU instructions), (3) the *action box*, i.e. abstractions for the non-blocking synchronization of active objects (i.e. threads), and (4) the *threads reel*, i.e. abstractions for the construction, scheduling, and destruction of active objects.

The system is implemented in C++ and runs (as guest level and in native mode) on i80x86-, i860-, sparc-, and ppc60x-based platforms. A port to C167-based, “CANned” μ -controllers is in progress. At the time being, the nucleus consists of 101 classes exporting 665 methods. Every class implements an *abstract data type*. Inheritance is employed extensively to build complex abstract data types. For example, the thread control block is made of about 45 classes arranged in a 11-level class hierarchy.

\$(PURPOSE)	scenario	size (in bytes)			
		<i>text</i>	<i>data</i>	<i>bss</i>	total
develop	null	10325	40	0	10365
	null (preemptive)	12933	44	0	12977
	signaller	10781	40	0	10821
	consumer/producer	11534	40	0	11574
	epilogue signalling	14024	60	0	14084
product	null	2226	40	0	2266
	null (preemptive)	3524	44	0	3568
	signaller	2466	40	0	2506
	consumer/producer	4003	40	0	4043
	epilogue signalling	4156	60	0	4216

Table 1: PURE memory consumption

As Table 1 shows, the highly modular nucleus structure still results in small and compact software. Two main PURE configurations are shown. The `develop` case stands for the procedure-based system representation in which every class serves as compilation/binding unit. In contrast, the `product` case stands for the macro-based system representation. In this case, class methods are the compilation/binding units. System generation time in the `product`-case is the 23-fold of the `develop`-case.

Five scenarios have been investigated, each one placing different demands on the underlying system in terms of functionality. The *null*-cases indicate the overhead to establish the initial active object (basing on cooperative and preemptive scheduling, resp.). In the case of *signaller*, an active object propagates signals to a signal box. Next, *consumer/producer* stands for a pair of dynamically created/destroyed active objects exchanging signals in a rendezvous-like manner. Finally, with *epilogue signalling*, an interrupt handler (so called epilogue) sends signals to and unblocks a waiting active object. The numbers were produced using GNU g++ 2.7.2.3 for the i586 running Linux Red Hat 5.0.

3 Concluding Remarks

In addition to providing a highly modular operating system for deeply embedded parallel/distributed systems, the PURE project can be seen also as an experiment in the exploitation of object orientation for the implementation of resource-sparing system software. The “old-fashioned” program-family concept has been applied consequently to create featherweight system abstractions. In addition, “standard” development tools have been used for system generation. The results

are quite promising, but not yet totally satisfactory.

Future work concentrates on the development of an operating-system *workbench*. The idea is to offer to system and even application programmers a set of tools that enable the construction, static/dynamic configuration, and (automatic) generation of application-oriented operating systems. The workbench will be evaluated using an object-oriented “workpiece” of OSEK [1], i.e. creating a “PURE-ly slim” but yet extensible family member for automotive systems.

References

- [1] OSEK/VDX Steering Committee. OSEK/VDX Operating System, October 1997. Version 2.0 revision 1.
- [2] A. N. Habermann, L. Flon, and L. Coopridier. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.
- [3] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *Transaction on Software Engineering*, SE-5(2), 1979.
- [4] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.
- [5] P. Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, 1986.