DESIGN RATIONALE OF THE PURE OBJECT-ORIENTED EMBEDDED OPERATING SYSTEM¹

Friedrich Schön*,
Wolfgang Schröder-Preikschat,
Olaf Spinczyk,
Ute 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

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. This paper discusses design issues of a family-based, object-oriented operating system targeting the area of deeply embedded systems in the above-mentioned sense. It is shown that a modular system consisting of a large number of reusable abstractions can be compiled to a fairly small and compact entity.

1. Introduction

Embedded systems are becoming more and more important --- and they are becoming more and more complex. Getting through daily life without being faced with electronically controlled devices is almost unthinkable. This holds not only for the general consumer market regarding cameras, HIFI, kitchen aids, washing machines etc., but also for other markets such as aircraft or automotive industries. Today's limousines, for example, can be considered (large scale) distributed systems on

¹ This work has been supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1.

wheels. There are cars in daily operation consisting of over 60 networked processors (i.e. μ -controllers). Although these systems are quite large with respect to the number of μ -controllers, they are still small with respect to memory size. In the above mentioned case, 1-2 MB of global memory (for the entire embedded distributed system) is not uncommon --- and this would already be more than luxurious. Typically, a single μ -controller-based *electronic control unit* (ECU) will be equipped with less than 4 KB of memory.

The complexity of these “decentralized computer architectures” can be managed no longer by the application alone. Dedicated embedded operating systems are required 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 describes the design concepts of the PURE embedded (distributed/parallel) operating system and discusses first implementation results. PURE is targeting deeply embedded systems, as e.g. automotive systems, and aims at providing a *portable, universal runtime executive* for resource-sparing (real-time) applications.

2. Design Approach

The approach followed by PURE is to understand an operating system as a *program family* (Parnas, 1979) and to use *object orientation* (Wegner, 1986) 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.

2.1 Incremental System Design

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* (Habermann, 1976), 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.

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.

2.2 Revival of Program Families

Applying the family concept in the software design process leads to a highly modular structure. Because of the strong analogy between the notions of “program family” and “object orientation” (Figure 1), it is almost natural to construct program families by using an object-oriented framework (Cordsen, 1991).

Figure 1. Program families vs. object orientation

Both approaches are, in a certain sense, dual to each other. The minimal subset of system functions in the program family concept has its counterpart in the superclass of the object-oriented approach. Minimal system extensions are thus introduced by means of subclassing. Inheritance and polymorphism are the proper mechanisms to allow different implementations of the same interface to coexist. Code reuse is significantly enhanced, increasing the commonalities of different family members.

In the realm of embedded distributed/parallel systems, inheritance must be applicable even in the case of crossing address space, node, and network boundaries (Nolte, 1998). This aspect is significant for the PURE development. The program family concept is the fundamental design principle. Object orientation is primarily being used as an implementation, not a design instrument.

2.3 Operating-System Construction Set

PURE is an open operating system. All its abstractions are revealed to system designers or even application programmers. The entire system is represented as a library, or a set of libraries, of small and “handy” object modules. These modules are small with respect to the number of exported references to functions or variables. This helps, e.g., state-of-the-art binders creating slim-line operating systems that contain only those components used (i.e. referenced) by a given application. Prerequisite however is a highly modular system architecture --- and this is achieved by a family-based design and an object-oriented implementation.

PURE has much in common with OSKit (Ford, 1997). Instead of inventing a new system architecture, as e.g. done with the Exokernel approach (Engler, 1995), PURE

provides abstractions that allows one to construct many of those architectures. An operating-system architecture is not prescribed by PURE. Rather, a construction set for the development of operating systems is established. Whether an operating system is monolithic or based, e.g., on micro-kernel technology, is up to the actual “mechanic” who uses PURE elements to create a product according to some blueprint. In order to create a tailor-made operating system, the blueprint comes from the application itself.

3. System Architecture

The coarse structure of PURE in addition to the correlation to some (maybe automotive control) application is depicted in Figure 2.

Figure 2. System architecture

PURE is made of a nucleus and nucleus extensions. The nucleus implements a *concurrent runtime executive*, i.e. CORE, for passive and active objects. By means of minimal *nucleus extensions*, NEXT, features such as application-oriented process and address space models, blocking (thread) synchronization, and problem-oriented (remote) message passing are added to the system. These extensions are present only if demanded by the application. They transform the nucleus into a distributed abstract thread processor.

3.1 The Nucleus

CORE implements a *minimal subset of system functions* for the scheduling of events and/or actions. Events are implemented by (hardware) interrupts, while actions are implemented by (featherweight) threads. The functions are provided by four building blocks:

1. the *flange*, i.e. abstractions for the attachment of objects to trap/interrupt vectors and propagation of 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 disabling of hardware interrupts),

3. the *signal-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.

Figure 3. Nucleus building blocks

Figure 3 shows a fifth building block, the (device) *driver*, which is to be taken into account in a complete system design. However, in PURE, device drivers are not really part of the nucleus, but of NEXT or even the application. Solely, CORE provides mechanisms for a smooth integration of driver modules. In addition to enabling the attachment/detachment of driver abstractions to/from CPU exception vector entries (supported by *flange*), functions are provided (by *sluice*) for the synchronization of interrupt-driven code. Drivers are considered a glue between non-synchronized and synchronized parts of a PURE-based system.

3.2 The Nucleus Family

Depending on the actual application requirements, the nucleus building blocks appear in different configurations. Each of these configurations represent a member of the nucleus family. An excerpt from the nucleus family tree is shown in Figure 4.

The picture indicates six family members, each of which implements a specific operating mode. Each member is built by one or more function blocks and described by a functional hierarchy of these blocks. The function blocks can be reused in various configurations. In order not to restrict the family, design decisions have been postponed as far as possible and encapsulated by higher-level abstractions. As a consequence, PURE can be customized, for example, with respect to the following scenarios:

1. One way of operating the CPU is to let PURE run *interruptedly*. This family member merely supports low-level trap/interrupt handling. The nucleus is free of any thread abstraction. It only provides means for attaching/detaching exception handlers to/from CPU exception vectors (*interruption*).

Figure 4. Nucleus family tree

2. In order to *reconcile* the asynchronously initiated actions of an *interrupt service routine* (ISR) with the synchronous execution of the interrupted program, a minimal extension to 1. was made. The originating family member ensures a synchronous operation of event handlers (*driving*) in an interrupt-transparent manner (*serialization*).
3. The second basic mode of operating the CPU means *exclusive* execution of a single active object. In this situation, the nucleus provides only means for *objectification* of a single thread. The entire system is under application control, whereby the application is assumed to appear as a specialized active object. There is only a single active object run by the system.
4. A minimal extension to 3. leads to *cooperative* thread scheduling. No other design decisions are made except that threads are implemented as active objects and scheduled entirely on behalf of the application (*threading*). There may be many active objects run by the system.
5. Adding support for the serialized execution of thread scheduling functions (*locking*) enables the *non-preemptive* processing of active objects in an interrupt-driven context. Thread scheduling still happens cooperatively, however the nucleus is prepared to schedule threads on behalf of application-level interrupt handlers. Actions of global significance, and enabled by interrupt handlers, are assumed to be synchronized properly (*serialization*).
6. Multiplexing the CPU between threads in an interrupt-driven manner establishes the autonomous, *preemptive* execution of active objects. In this

case, NEXT extends the nucleus by a device driver module (*multiplexing*) taking care of timed thread scheduling.

Comparing Figure 4 to Figures 2 and 3, the flange takes care of interruption, sluice is responsible for serialization, the threads reel implements locking, threading, and objectification, and NEXT covers (device) driving and (CPU) multiplexing.

3.3 Interrupt Transparency

A crucial aspect in the design of (embedded) operating systems concerns interrupt handling, in particular the synchronization of asynchronously initiated execution of code fractions. The underlying model has to be efficient and should also reduce the periods of times during which interrupts are physically disabled to an absolute minimum. Ideally, at software level, interrupts should never be disabled at all. The PURE concept follows this ideal pattern and, thus, ensures *interrupt transparency*: at no time, PURE disables (hardware) interrupts, or exploits special CPU instructions, for synchronization purposes.

In order to safely execute interrupt-driven concurrent code, a PURE-based interrupt handling subsystem, such as a device driver, is to be structured into two parts. The first part, the *prologue*, is attached to some CPU exception vector and will be immediately started upon signaling a hardware interrupt to the CPU. The second part, the *epilogue*, is considered a prologue continuation and will be started only if critical sections are inactive. Prologues always execute with the actual hardware priority. Except in the case of non-maskable interrupts (NMI), a high-priority prologue thus may prevent the invocation of a lower (or same) priority prologue. In contrast, epilogues always execute with all interrupts enabled, similar to the NMI case.

Prologues and epilogues are (logically) loosely coupled by a *guard* who takes care of the invocation of pending epilogues at the right moment. When a prologue wishes to start a continuation, it hands over to the guard the epilogue object to be synchronized/serialized. If the guard is free, the epilogue is called without any further delay. Otherwise, the epilogue is made pending and will be scheduled when the guard becomes free again.

A guard behaves like a *semaphore*, except that the delayed objects are not active (i.e. processes) but passive. The currently running, but interrupted, thread will never block when epilogues are to be made pending. The guard maintains a queue of epilogue objects. Since the guard operates in an interrupt-driven context, its queue operations need to be synchronized. This happens in an interrupt-transparent manner. A queuing strategy has been implemented that enables the dequeue operation, in order to run delayed epilogues upon leaving a guarded section, to be overlapped by enqueue operations issued by an interrupt handler (i.e. prologue). In addition, enqueue operations are allowed to overlap themselves, e.g. when a high-priority interrupt handler preempts a low-priority interrupt handler. In contrast to the PURE predecessor PEACE (Schröder-Preikschat, 1994), all necessary synchronization measures run with interrupts enabled. In PURE, the actual interrupt priority of a running process will never be defined by software, but only by the hardware.

4. Aralysis

The PURE 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 C 167-based, “CANned” μ -controllers is in progress. At the time being, the nucleus consists of over 100 classes exporting over 600 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 14-level class hierarchy. Thus, PURE is of really high modular structure. This rises the question of costs in terms of memory consumption and performance.

4.1 Memory Consumption

As Table 1 shows, the highly modular nucleus structure still results in a small and compact implementation. Basis are the members of the nucleus family tree described along with Figure 4. The numbers were produced using GNU g++ 2.7.2.3 for the i586 running Linux Red Hat 5.0.

family member	size (in bytes)			
	text	data	bss	total
interruptedly	812	64	392	1268
serialize	1882	8	416	2306
exclusive	434	0	0	434
cooperative	1620	0	28	1648
non-preemptive	1671	0	28	1699
preemptive	3642	8	428	4062

Table 1. PURE memory consumption

A PURE implementation is represented in two different ways. In the development phase, a procedure-based system representation founds the basis. In this case, every class serves as compilation/binding unit. In the prototyping phase, a macro-based system representation is used. In this case, class methods are the compilation/binding units. Compared to a PURE development system, memory consumption of a PURE prototype is about 3-7 times smaller, but system generation time can be the 23-fold.

4.2 Performance

Figure 6 shows the number of (i586) CPU clock cycles spent during interrupt processing. The clock frequency in this experiment was 166 MHz and the measurement was made reading the i586 (on-chip) counter register. Again, the above-mentioned C++ compiler was used.

Figure 6. Interrupt latency

At clock 0, a thread will be interrupted. The numbers show that, depending on whether a guard is free (*lock free*) or locked (*lock set*), normal thread execution will be suspended for about 238 resp. 261 clock cycles. Prologue execution starts with clock tick 79 resp. 87. The minor difference is due to the different program instrumentations for the lock-free and lock-set cases. Epilogue execution starts with clock cycle 147 resp. 231. The difference of 84 cycles indicates the best-case delay of an epilogue when the guard is locked. Further delays depend on two factors: (1) the time to *leave* the guarded section² and (2) the number of preceding epilogues in the queue. Delaying an epilogue takes 32 clock cycles and propagating a delayed epilogue takes 65 clock cycles. Both phases pass through the interrupt-transparent queue synchronization. Resuming an interrupted thread when a prologue terminates costs 47 clock cycles. This is independent of whether the thread was interrupted inside or outside a guarded section. Leaving an epilogue takes 44 resp. 30 clock cycles. The difference is due to the different actions to be taken when returning from a lock-free or a lock-set case.

The numbers for thread scheduling depend on the employed nucleus family member (refer to Figure 4). They range from 49 clock cycles (*cooperative*) to 300 clock cycles (*preemptive*). All these performance figures demonstrate the still “featherweight” structure of PURE, although quite a large amount of abstractions (classes, modules, functions) are involved in all the occurrences. But: “*It is the system design which is hierarchical, not its implementation*” (Habermann, 1976).

5. Conclusion

Deeply embedded operating systems have their own *raison d'être*. They are designed to specifically support the execution of (control) applications under extremely limited resource constraints. In consideration of the specific demands of these applications, it becomes extremely difficult, if not impossible, successfully adopting existing operating systems.

In the realm of embedded systems consisting of several tens of interconnected (autonomous) μ -controllers, the operating-system design is crucial to meeting the application demands. A lightweight system structure is required, meaning to advance into traditional frontier districts of operating systems. The boundary where an

² The worst-case execution path of a guarded section has not yet been determined. However, it will be in the order of 300 cycles, which relates to the overhead of timer-driven thread rescheduling.

embedded operating system begins and a runtime environment ends becomes indistinct. Furthermore, the boundary between application and operating system becomes indistinct. Distributed (parallel) embedded systems make the amalgamation of hardware, operating systems, runtime environments, compiler, programming language and application more important than ever.

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 the OSEK operating system (OSEK, 1997), i.e. creating a “PURE-ly slim” but yet extensible family member for automotive systems.

References

- OSEK/VDX Steering Committee (1997). OSEK/VDX Operating System, October 1997. Version 2.0 revision 1.
- Cordsen, J. and Schröder-Preikschat, W. (1991). “Object-Oriented Operating System Design and the Revival of Program Families.” In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (I-WOOS '91)*, pages 24-28, Palo Alto, CA, October 17-18.
- Engler, D. R. and Kaashoek, M. F. and O'Toole Jr., J. (1995) “Exokernel: An Operating System Architecture for Application-Level Resource Management.” In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 251-266, Copper Mountain Resort, Colorado.
- Ford, B. and Back, G. and Benson, G. and Lepreau, J. and Lin, A. and Shivers, O. (1997). “The Flux OSKit: A Substrate for Kernel and Language Research.” In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 38-51, Saint-Malo, France.
- Habermann, A. N. and Flon, L. and Coopridner, L. (1976). “Modularization and Hierarchy in a Family of Operating Systems.” *Communications of the ACM*, 19(5):266-272.
- Nolte, J. and Schröder-Preikschat, W. (1998). “Dual Objects -- An Object Model for Distributed System Programming.” In *Proceedings of the Eighth ACM SIGOPS European Workshop, Support for Composing Distributed Applications*, <http://www.acm.org/sigops/EW98/papers.html>.
- Parnas, D. L. (1979). “Designing Software for Ease of Extension and Contraction.” *Transaction on Software Engineering*, SE-5(2).
- Schröder-Preikschat, W. (1994). “The Logical Design of Parallel Operating Systems.” *Prentice Hall International*, ISBN 0-13-183369-3.
- Wegner, P. (1986). “Classification in Object-Oriented Systems.” *SIGPLAN Notices*, 21(10):173-182.