An Aspect-Oriented Implementation of Interrupt Synchronization in the PURE Operating System Family*

Daniel Mahrenholz† Olaf Spinczyk† Andreas Gal‡ Wolfgang Schröder-Preikschat†

†University of Magdeburg
Universitätsplatz 2
39106 Magdeburg, Germany
{mahrenho,olaf,wosch}@ivs.cs.uni-magdeburg.de

‡University of California, Irvine
Dept. of Information and Computer Science
Irvine, CA 92697-3425, USA
gal@uci.edu

Abstract

Some concerns in the design of operating systems are hard to modularize in the implementation and thus difficult to maintain. One of these "crosscutting concerns" is the interrupt synchronization strategy. Changing that strategy is typically expensive and risky. Aspect-oriented programming (AOP) is a promising approach to overcome these problems, but most aspect-oriented programming languages are not adequate for the operating systems domain. Thus experiences with AOP and operating systems are rare.

Here we describe our experiences with an aspect-oriented implementation of interrupt synchronization in the PURE *operating system family using AspectC++ [1], a new aspect-oriented language extension for C++ designed by the authors. We provide a critical evaluation of our new approach comparing it to the previous non aspect-oriented implementation and prove that AOP does not impose an unacceptable overhead.*

1 Introduction

Aspect-oriented programming (AOP) [4] has found so far only limited application in the field of operating system design and implementation. This cannot only be explained with the fact that operating system implementors are traditionally very conservative with their choice of implementation language, design approaches, and tools, but rather with the limited availability of aspect-oriented languages and supporting tools.

This has changed with the public availability of AspectC++ [7], an aspect-oriented language extension for C++ [8]. The AspectC++ compiler is implemented as a C++ front-end, allowing the operating system code to be compiled to native machine code with any (optimizing) C++ compiler. At the same time AspectC++ requires no additional run-time system, making it an ideal language for aspect and object-oriented operating system development.

This paper presents our experiences with aspect-orientation and operating systems as a case study: the aspect-oriented reimplementation of interrupt synchronization in the PURE operating system family [1].

The interrupt synchronization strategy directly affects the overall system performance and response time, and is closely related to the system architecture. It is implemented by placing calls to synchronization primitives into the operating system code. The placement depends on the strategy, which can be fine-grained, coarse-grained, or something in between. With traditional techniques the implementation is hard to modularize and modifying the strategy always becomes a risky and expensive effort. Thus, interrupt synchronization can be classified as a *crosscutting concern*. It is a good example to study the benefits of AOP in the operating system domain.

The case study shows how AOP can be applied to support the modular implementation of the interrupt synchronization strategy leading to code that is easier to develop and maintain without any unacceptable overhead.

The remaining sections of this paper are structured as follows. Section 2 gives a brief introduction into the concept of aspect-oriented programming. It is followed by Section 3, which describes the most important language elements of AspectC++. Section 4 presents our case study. It introduces the interrupt synchronization model of PURE, its old implementation, and the new implementation after reengineering this part using aspects. The paper ends with a discussion of related work in Section 5 and our conclusions

*This work has been partly supported by the German Research Council (DFG), grant no. SCHR 603/1 and SCHR 603/2 and the National Science Foundation (NSF) under grants EIA-9975053 and CCR-0105710.

[1] AspectC++ is freely available from http://aspectc.org/

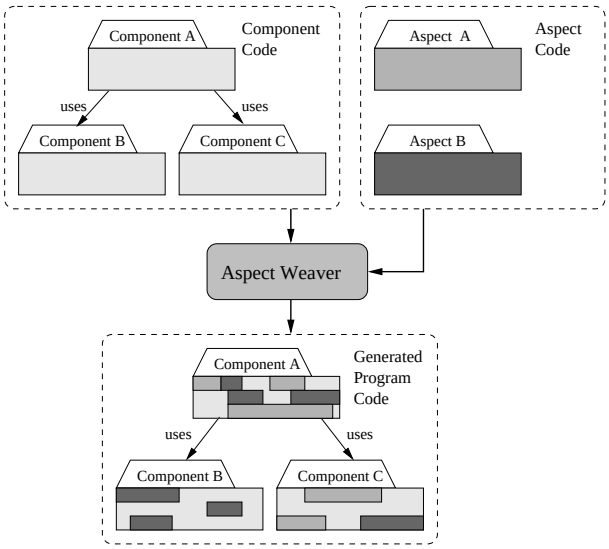

Figure 1. Weaving aspects

as well as future work in Section 6.

2 Aspect-Oriented Programming

Aspect-oriented programming tries to solve the problem that often a single dimension of functional decomposition is not sufficient to implement all aspects of a program in a modular way. This means that code stemming from a single design decision is widely spread over the system. It cannot be encapsulated in a single function, class or module. This so called *aspect code* is tangled with the functional *component code* that fits into the functional decomposition scheme. A widely used example for this effect is the synchronization code in non-sequential programs like operating systems. While a single strategic design decision dictates at which points synchronization primitives must be invoked, the implementation of this decision cannot be modularized using traditional approaches.

AOP helps out of this dilemma, because an aspect-oriented language environment allows to implement such crosscutting concerns in modular units called *aspects*. The aspect code guides a tool, the *aspect weaver*, in inserting code fragments specified by the aspect code at locations where they are required. These insertion points are called *join points*. This allows the synchronization strategy implementation to be separated from the operating system components on the source code level. As a result, both are better to reuse and maintain.

Aspect weaving can be performed at compile-time or at run-time. This paper focuses on *compile-time aspect weaving*, because the additional cost for having an aspect weaver in the target system performing run-time code injection is not acceptable where highest efficiency must be provided.

3 AspectC++

AspectC++ is an extension to the C/C++ programming language. Its aim is to support aspect-oriented programming even in domains where resource limitations do not allow to use more expensive languages like AspectJ[2], which requires a Java run-time environment.

This section gives an overview of the key concepts of this new language. A more in-depth introduction to AspectC++ can be found in [7].

Join Points are points in the component code where aspects can interfere. A join point refers to a method, an attribute, a type, an object, or a point in the control flow from which a join point is accessed (e.g. a method call).

Pointcuts are the key language element to deal with the crosscutting nature of aspects. They can describe points in the static structure or the dynamic control flow of a program in a highly flexible manner. Technically, pointcuts are sets of join points described by a *pointcut expression*. Pointcut expressions are composed from *match expressions* used to find a set of join points, *pointcut functions* used to filter or map specific join points from a pointcut, and algebraic operators used to combine pointcuts.

Advice definitions can be used to specify code that should run when the join points specified by a pointcut expression are reached. Different kinds of advice can be declared, including *after* advice that runs after the join point, *before* advice that is executed before the join point, and *around* advice, which is executed in place of the join point.

Introductions can add certain items to the static program structure. For example introductions can add new methods, attributes, or base classes to join points of type “class” contained in a pointcut expression.

Aspects implement in a modular way crosscutting concerns and are an extension to the class concept of C++. Additionally to attributes and methods, aspects may also contain advice declarations. The advice code can use the attributes stored in an aspect instance to preserve state information between different invocations. AspectC++ offers virtual pointcuts and aspect inheritance to support the reuse of aspects. A virtual pointcut can be redefined in a derived aspect and the inherited advice will use the new pointcut definition. Pointcuts

[2] `http://aspectj.org/`

can also be pure virtual. In this case the pointcut definition has to be overwritten by a derived aspect before it can be instantiated. Aspects can also inherit from a class, but it is not possible to derive a class from an aspect.

4 Case Study: Interrupt Synchronization

In this section we will take a closer look at interrupt synchronization in the PURE operating system family. We first present the pro-/epilogue synchronization model, which is used in most PURE configurations, and the status quo of the implementation. Later an aspect-oriented reimplementation of the synchronization strategy is presented in two variants. Finally, the memory consumption in terms of code size for all three implementations is compared to evaluate the eventual overhead of the aspect-oriented implementation.

4.1 Pro-/Epilogue Model

Interrupt handling is one of the most important tasks of an operating system. In PURE this task is accomplished using a pro-/epilogue model [6]. In this model the kernel is synchronized using a single lock variable. This lock is set when a thread enters the kernel and cleared when it leaves it. Interrupt service routines cannot be allowed to make calls to the kernel during this time to avoid corruption of kernel data structures. Therefore, interrupt service routines are split into two parts: the *prologue* and the *epilogue*. The prologue contains the instructions for a fast response to the device interrupt and the epilogue contains all operations that interact with the kernel. The epilogue is not called directly but through a special class: the *guard*. The task of the guard is to check the lock variable before executing the epilogue. In the case of a locked kernel the execution of the epilogue is postponed until the lock is released. Otherwise the epilogue is executed immediately.

4.2 Status Quo

In a PURE system the global lock variable is set and cleared using `lock.enter()` resp. `lock.leave()`. An analysis of the source showed 166 different calls to these methods spread out over 15 classes. Figure 2 shows a part of the PURE class hierarchy including 8 of the 15 mentioned classes (marked with a bold line and a border). It is obvious that interrupt synchronization is a highly crosscutting concern, because it affects many classes in different subsystems.

The class diagram reflects the layered design of PURE: each (often configurable) layer is implemented as a new level in the inheritance tree. Calls to the synchronization primitives are regarded as an extension in their own layer

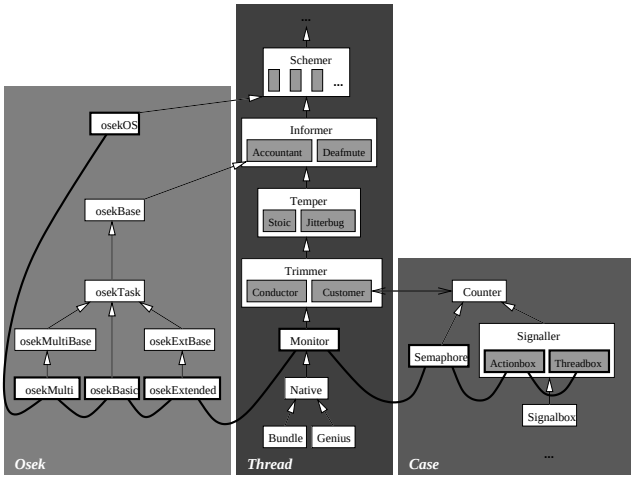

Figure 2. Access to the global lock variable

and thus often are implemented in their own class (e.g. `Monitor`).

This implementation of interrupt synchronization causes some problems, which PURE might share with other object-oriented operating systems:

- Many subsystems (here *osek*, *thread*, *case*) depend on the implementation (or at least the interface) of the interrupt synchronization primitives. Therefore, they can't be easily reused in other contexts or system configurations that use a different synchronization scheme (or none at all) without error-prone manual code manipulation. Subsystems cannot become truly reusable components.

- The layer, which adds the synchronization code, also automatically defines the API for the application programmer. While the application is allowed to use derived classes, base classes of the API must not be called as this would break the defined synchronization scheme. This API restriction makes it impossible to implement application-defined threads derived directly from lower-level thread abstractions.

- A similar problem exists for operating system developers. If they extend the system by deriving from already synchronized classes, the added code has to be synchronized separately. This means that the lock has to be released before calling methods of the synchronized base class to retain their calling semantic and to avoid misbehavior. However, such an unlocked period, even if it is very short, is not acceptable in certain situations. The only way to circumvent this problem is to adapt the synchronization layer, which once again results in a lot of manual intervention.

All these problems with the current PURE implementation show that interrupt synchronization (especially finding synchronization points) is a difficult task. This task should not be burdened on developers of single operating system components. The placement of locks and their actual implementation should better be regarded as a strategic decision, because of its influence on the whole system. Following the concept of *separation of concerns* this decision should be made independently. Global system requirements must be considered, which the implementor of a single system component, e.g. a device driver, does not and should not know. Interrupt latencies are a good example for such requirements. With a fine-grained locking strategy they can be made short at the cost of some run-time and code overhead. A coarse-grained strategy in contrast can reduce the run-time overhead simply because it needs less lock/unlock calls. This often results in higher interrupt latencies.

4.3 Variant 1

In our first example we pursue a naive approach to separate the interrupt synchronization strategy implementation from the component code. The aspect (Figure 3) works straightforward. A pointcut `locked` is defined by using a number of match expressions to specify exactly the set of methods that should be executed synchronized. Likewise, we define the pointcuts `enter` and `leave` for some special functions that set or resp. clear the lock. The pointcut `upcall` describes the functions, which leave the synchronized system part. These functions have to clear the lock before leaving the kernel and set it afterwards. The advice definition in line 17 specifies that `lock.enter()` should be called before any of the join points from the pointcuts `locked` or `enter` is reached. In our example this means: before any of the selected functions is executed. The advice definition in line 18 specifies that `lock.leave()` should be called after the given join points, i.e. after executing the selected functions. In line 19-20 we do the inverse for all functions that temporary leave the kernel.

This variant of the aspect has a major problem: it is not robust. After each change in the PURE system the aspect code has to be revised to make sure that the functions selected for the different pointcuts still match the actual system.

4.4 Variant 2

The second variant solves this weakness as it works on a higher level of abstraction. It first defines the pointcuts `thread`, `sched`, `ipc`, `guard`, `driver`, and `mm`. Each of these pointcuts corresponds to a PURE subsystem (Figure 4). The additional pointcut `api_layer` describes the PURE API, i.e. all classes that are available for application programmers.

The idea behind Variant 2 is that locking primitives should be called whenever the subsystem boundary of a synchronized region is crossed. The definition of the pointcut `locked` in line 6 of Figure 5 shows this clearly. The `call` pointcut function provides all call join points where the target function is a member of one of the specified classes. Here the specified classes are all classes that belong to the synchronized kernel region. From all these only those calls are selected that are not located within a kernel class. This means a call from outside into the synchronized kernel region.

With this variant it is still necessary to provide names. However, with pointcut functions like `derived` only the most important base classes must be listed and there is no problem with the robustness in case of extensions. Using namespaces would further simplify the component definitions.

example	**system**	**size** (in bytes)			
		text	data	bss	total
friend	status quo	4724	576	80	5380
	variant 1	4424	612	80	5116
	variant 2	4248	540	80	4868
philo	status quo	8438	948	108	9494
	variant 1	8470	984	108	9562
	variant 2	8294	952	108	9344
clock	status quo	9731	1124	768	11623
	variant 1	9939	1144	772	11855
	variant 2	9699	1128	736	11563

Table 1. Comparison of memory consumption

4.5 Comparison

Table 1 shows a comparison of the code sizes [3] for the old implementation ("status quo"), Variant 1, and Variant 2. PURE systems aim at supporting applications in the area of deeply embedded systems and thus are configured for and linked with the application. This means that the size strongly depends on the application. We have therefore measured three test applications: **friend** uses cooperative threads and no interrupts while **philo** and **clock** use preemptive interrupt driven thread scheduling. Threads used by **philo** are based on the class `Native` and **clock** uses the thread abstractions of the *osek* subsystem (see Figure 2).

The measurements show that the aspect-oriented implementations do not increase the code size of our systems. The opposite is true: especially for Variant 2 a significant

[3] Linux/x86 guest level implementation, compiled with gcc 2.96

```
aspect IntSync1
 {
   pointcut thread () = within ("osekExpanded"||"osekPlain"||"osekSeveral"||"Monitor");

   pointcut locked () =
      (execution ("% Actionbox::%(...)") && !execution ("% %::unload()")) ||
      execution ("% BeamerGuard::%(...)") || execution ("% RateGuard::%(...)") ||
      (execution ("% Semaphore::%(...)") && !execution ("% %::p()") && !execution ("% %::v()")) ||
      execution ("% Threadbox::%(...)") ||
      ((thread () && execution ("% %::%(...)")) &&
       !execution ("% %::hello()") && !execution ("% %::liedown()") &&
       !execution ("% %::getup()") && !execution ("% %::operator ()()")) ||
      execution ("% operator new(...)") || execution ("% operator new[](...)") ||
      execution ("% operator delete(...)") || execution ("% operator delete[](...)");

   pointcut upcall() = within("Actionbox") && call ("% Action::action()");
   pointcut enter () = thread () && execution ("% %::getup()") || execution ("% osekOS::start(...)");
   pointcut leave () = thread () && (execution ("% %::hello()") || execution ("% %::liedown()"));

   advice locked () || enter () : before () { lock.enter (); }
   advice locked () || leave () : after ()  { lock.leave (); }
   advice upcall () : before () { lock.leave (); }
   advice upcall () : after ()  { lock.enter (); }
 };
```

Figure 3. Variant 1: Aspect Definition

```
// The main PURE components: threads, the scheduler, IPC classes, ...
pointcut api_layer () = classes ("PURE::%");
pointcut thread ()    = (derived ("Entrant") || classes ("SeqPED%"|| "%Context"||
                         "osekChainStore"||"osekMultiStore")) && !api_layer ();
pointcut sched ()     = (derived ("TaskQueue") || classes ("Contest")) && !api_layer ();
pointcut ipc ()       = (derived ("Counter"||"Lineup"||"Booster"||"Alarm"||"Linkage") ||
                         classes ("Patient"||"Impatient")) && !api_layer ();
pointcut guard ()     = classes ("Date"||"Mediator");
pointcut driver ()    = derived ("Date") && !guard () && !api_layer ();
pointcut mm ()        = classes ("%Economist"||"Pile"||"Registers") && !api_layer ();

// Kernel entry points
pointcut enter () = execution ("% osekOS::start(...)");

// Upcalls that leave the kernel area or points where the lock must be released
pointcut leave() =  call ("void Triplet::action()") || (execution ("void %::sleep(...)") &&
                    within (sched ()));
```

Figure 4. Variant 2: Component Definition

```
aspect IntSync2
 {
   // coarse-grained locking strategy: all these parts are locked with a single guard
   pointcut kernel () = thread () || ipc () || sched () || driver () || mm ();

   // lock all calls into the kernel from outside and unlock when the kernel is left
   pointcut locked ()   = (call (kernel ()) && !within (kernel ())) || enter ();
   pointcut unlocked () = leave ();

   // the advice is simple now:
   advice locked ()   : before () { lock.enter (); }
   advice locked ()   : after ()  { lock.leave (); }
   advice unlocked () : before () { lock.leave (); }
   advice unlocked () : after ()  { lock.enter (); }
 };
```

Figure 5. Variant 2: Aspect Definition

reduction in code size can be observed. The reason for this is that we were able to remove classes from the inheritance trees, which were only used to call synchronization primitives before and after calling an overridden base class function. Of course, the synchronization calls did not vanish. Instead, AspectC++ generates them mixed with the code of other system layers, resulting in more compact code.

5 Related Work

As we have already mentioned earlier, the application of AOP in the operating systems domain has not been studied intensively yet.

An interesting approach to reconcile a traditionally designed and implemented operating system with AOP is pursued by Coady et al [3, 2]. In these two case studies the authors use AspectC [4] to re-engineer parts of the FreeBSD v3.3[5] operating system exploiting the advantages of aspect-oriented programming. In particular, the first case study [3] implements the prefetching strategy of the filesystem in a modular way using aspects. The second case study [2] implements the prefetching and write cache strategies of the Network File System (NFS) in a similar way in order to show that AOP can assist in realizing an extensible design.

Another approach is pursued by the AOSA project [5], which tries to support AOP in operating systems without the need for a special language support. The compile-time aspect weaving is replaced with an *aspect moderator*, a special component in the system that dispatches all method calls and redirects them to the aspects, which are implemented as regular C++ classes. This approach allows only for relatively simple run-time aspect weaving, but is at the same time relatively easy to implement and allows to add and remove aspects dynamically.

All the mentioned papers give an idea about the potential of AOP. However, it remained unclear whether there are any code size or performance penalties.

6 Conclusions and Future Work

With our case study we were able to show how a crosscutting concern in operating system code like the interrupt synchronization strategy can be implemented in a modular way. This allows to implement changes to this strategy with minimal costs and is a step towards a truly component based operating system.

For the implementation work we used AspectC++, an aspect-oriented language extension for C++. The measurements of code sizes prove that the increased modularity does not necessarily mean extra costs at run-time. In some cases the AspectC++ code was even more compact. This result is very encouraging and we will continue our research with other crosscutting concerns in operating system code.

A lesson we have learned during this work is that aspects and component code should be designed together. Aspects require pointcut definitions. If the component code was not designed appropriately, these definitions sometimes consist of hundreds of function names, eventually causing massive maintenance problems. While integrated development environments with knowledge about the aspect concept could possibly reduce the impact of this problem, a “design for aspect intervention” can prevent it.

One area of future work is to explore the possibility how run-time aspect weaving could be used to allow operating systems to dynamically adapt global strategies to load situations.

[4] `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`

[5] `http://www.freebsd.org/`

References

[1] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The Pure Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *IEEE Proceedings ISORC'99*, 1999.

[2] Y. Coady, A. Brodsky, D. Brodsky, J. Pomkoski, S. Gudmundson, J. S. Ong, and G. Kiczales. Can AOP Support Extensibility in Client-Server Architectures? In *European Conference on Object-Oriented Programming (ECOOP), Aspect-Oriented Programming Workshop*, June 2001.

[3] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring Operating System Aspects. *Communications of the ACM*, pages 79–82, Oct. 2001.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[5] P. Netinant, C. A. Constantinides, A. Bader, and T. Elrad. Supporting the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks. In *International Conference on Parallel and Distributed Techniques and Applications (PDPTA 2000)*, Las Vegas, Nevada, June 2000. special session on Aspect-Oriented Programming.

[6] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[7] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of *Conferences in Research and Practice in Information Technology*. ACS, 2002.

[8] B. Stroustrup. *The C++ Programming Language — Second Edition*. Addison-Wesley, 1991.