

Programming OS Schedulers with Domain-Specific Languages and Aspects: New Approaches for OS Kernel Engineering*

Luciano Porto Barreto

COMPOSE group, <http://compose.labri.fr/>
INRIA/LaBRI/Enseirb, 33405 Talence Cedex, France
Luciano.Barreto@labri.fr

Rémi Douence, Gilles Muller, Mario Südholt

École des Mines de Nantes
4, rue Alfred Kastler, La chantrerie, Nantes
{douence,gmuller,sudholt}@emn.fr

Abstract

There is a continuous demand for new scheduling policies to address specific requirements of modern OSes. However, the implementation of such policies within an existing OS kernel raises many problems, mainly because optimizations within schedulers hinder code maintenance and implementation of existing schedulers is spread over the kernel.

In this paper we motivate that schedulers form an aspect within OS kernels. We show how the DSL of the Bossa system for the definition of scheduling policies and its runtime support can be integrated with a framework for Aspect-Oriented Programming, Event-based AOP. Finally, we discuss the generalization of AOP-based techniques to other OS kernel modules.

1 Introduction

Over the recent years, there has been a continuous demand for new scheduling policies to address specific requirements of modern OSes and emerging applications. Examples include policies for multimedia and real-time applications [2, 3, 6, 10, 14] and energy-based policies so as to increase the mission time of portable devices [8, 12, 11].

While the need for new scheduling policies is well recognized, their implementation within an existing OS kernel raises many problems. Based on an analysis of several OS kernels such as RT-Linux, Linux and BSD, we have identified the following difficulties in integrating a new scheduling policy:

- **Massive optimizations hinder code maintenance.**

Because schedulers are executed very frequently, they should be highly optimized so as to not

degrade overall system performance. In fact, critical parts of schedulers are often written in assembly code and meticulously structured to exploit specific features of the target architecture. As a consequence, the scheduling policy is mixed with low-level optimizations, thus obfuscating the implementation and complicating the development of new policies. Additionally, architecture-dependent optimizations that were initially valid can be useless or even degrade performance on the next of generation processors.

- **Implementation of existing schedulers is spread over the kernel.** A scheduler is often tied to multiple kernel mechanisms such as process synchronization, system calls, and device drivers. As such, it is common to find scheduling-related code spread over different parts of the kernel [13]. Only few experts are able to fully understand how the scheduler really works, even in well-documented OSes such as Linux and BSD.

These problems discourage real experimentation on OSes and restrain a wide dissemination of research results. In industry, the situation is even worse since developers have tight time-to-market constraints. Therefore, developers have little time to devote to risky tasks such as implementing a new scheduling policy.

Such an engineering nightmare calls for the use of new methodologies that can improve the implementation of OS kernels. In this paper we discuss the use of two approaches, Domain-Specific Languages (DSLs) and Aspect-Oriented Programming (AOP) that are promising in engineering operating system kernels.

A DSL is a high-level language providing constructs appropriate to a particular class of problems. The use of such a language simplifies programming, because solutions can be expressed in a way that is natural to the domain and because low-level optimiza-

*This work has been partially funded by the EU project "Easy-Comp" (www.easycomp.org), no. IST-1999014191

tions and domain expertise are captured in the language implementation rather than being coded explicitly by the programmer [9]. Recently, Barreto and Muller have presented Bossa, a DSL for programming schedulers [1]. This language simplifies scheduler programming and allows the verification of critical safety properties of a scheduler at compile time

Aspect-Oriented Programming [7] has recently been introduced to address problems involving code tangling, i.e. the implementation of concepts which cannot be encapsulated using a given programming paradigm and are scattered all over a program. Technically, AOP is aiming at language support for the elimination of tangling and is looking for corresponding translation techniques, commonly called code weaving. Recently, Coady *et al.* have demonstrated the benefit of aspects in OS implementation by adapting the cache behavior in the BSD file systems [4]. We discuss how Bossa’s approach to scheduler definition can be integrated into an aspect-oriented approach.

The rest of the paper is organized as follows: Section 2 presents Bossa and the benefits of using a DSL for programming schedulers. Section 3 discusses that scheduling can advantageously be treated as an aspect and how this can be done. Section 4 presents a perspective: how to generalize such techniques to an AOP-based OS kernel.

2 Bossa: a DSL for programming schedulers

Bossa has been designed to address two goals: (i) to evolve the scheduler into a modular kernel component; this is achieved by reengineering the kernel around an event-based run-time system, (ii) to ease the development of scheduling policies and to make possible the verification of important safety properties that are specific to the domain of scheduling; this is achieved by the Bossa DSL. We now highlight the main characteristics of Bossa [1].¹

2.1 The Bossa run-time system

Evolving the scheduler into a modular component requires substantial kernel reengineering. First, code fragments related to scheduling, i.e. *scheduling points*, that are initially spread over the kernel must be carefully identified. Then, scheduling points are replaced by a corresponding event notification to the Bossa Run-Time System (RTS). For that, the RTS provides a set of events to identify scheduling points (see Table 1). There are events for signaling process

creation, process termination, process blocking and unblocking. Events are organized as a hierarchy that indicates the subsystem or the driver that is the source of the event. This permits an event handler to treat either a generic case (i.e., all blocking events) or specific instances (i.e., blocking events from the `scsi` disk driver). Finally, an event handler has to be written for each event to be considered. All event handlers are centralized in a single module, i.e. the scheduling policy. As a result, changing or evolving the scheduling policy amounts to modification of only one module.

Table 1: Bossa events

Event	Generated by
<code>processBlock.*</code>	I/O calls, drivers
<code>processUnblock.*</code>	drivers, time service
<code>processYield</code>	<code>sched_yield()</code> primitive
<code>clockTick</code>	Clock interrupt handler
<code>processNew</code>	<code>fork()</code> sys. call: <code>clone()</code> , <code>exec*()</code>
<code>processEnd</code>	<code>exit()</code> , <code>kill()</code>
<code>Schedule</code>	Bossa run-time system

The Bossa RTS has been implemented within the Linux kernel. Initial performance evaluations show that Bossa induces an overhead of below 5%. We are currently experimenting with other schedulers such as a variant of BSD.

2.2 The Bossa DSL

The Bossa language provides high level abstractions such as process attributes, process states, process lists and events. A Bossa scheduler contains three parts: process state and attribute declaration, event handler definition, and an interface for interaction with processes managed by the scheduler. The Bossa compiler translates a Bossa scheduler into a C module that can be linked with the kernel and the RTS.

We now introduce the main features of the Bossa DSL by presenting code excerpts of the Bossa implementation of a simple priority-based scheduler.

Declarations

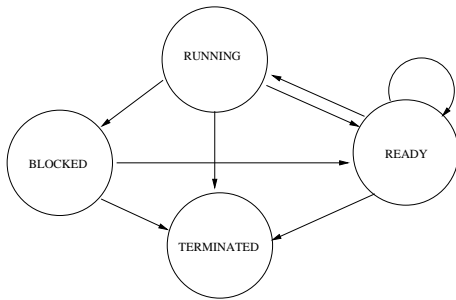
A scheduler defines a `process` type that contains attributes to support policy execution. In the priority-based scheduler, `process` only contains an integer that defines the process priority:

```
process = { int priority; };
```

A process managed by a Bossa scheduler is associated with a state that describes its current activity. A process can be either in a `RUNNING`, `READY`

¹Policy examples and a complete definition of Bossa is available at <http://compose.labri.fr/prototypes/bossa>.

or `BLOCKED` state. Additionally, a `TERMINATED` pseudo state is associated with a process that finishes. Only specific state transitions are allowed as illustrated by the following automaton:



To elect or preempt a process, it is necessary to be able to compare two processes according to a scheduling-specific relation. In Bossa, the ordering of processes is specified by the `ordering_criteria` declaration. The excerpt below specifies that the priority-based scheduler always selects the process with the greater `priority` attribute.

```
ordering_criteria = {highest priority};
```

When designing a policy, it is necessary to define support for storing processes. For that need, Bossa provides process queues and process variables. Queues and variables are always associated with a state; several queues and variables may be associated with a single state. Indeed, queues and variables refine the basic state abstraction for a specific policy.

The excerpt below specifies that there is only one process in the `RUNNING` state (i.e., running on the processor), that there is a list of processes associated with the `READY` state which is sorted according to the previous `ordering_criteria`, that there is a `fifo` list of processes associated with the `BLOCKED` state, and a process in `TERMINATED`. (Note that no storing support is associated with `TERMINATED`.)

```
states = {
  RUNNING running : process;
  READY ready    : sorted queue;
  BLOCKED blocked : fifo queue;
  TERMINATED terminated;
};
```

Event handlers

The behavior of a scheduler is determined by the events it subscribes to. For each event, the policy must define an event handler that specifies the actions to be executed when the event occurs. An event handler basically updates process attributes and performs state changes (which are denoted by the move operator `=>`).

The following code excerpt specifies the behavior of the scheduler when a process unblocks. First, the

scheduler moves the process for which the event was generated to the `ready` state. If the process that unblocks (`e.target`) has a greater priority than the running process using the `>` operator, the scheduler preempts the running process (by moving it back to the `ready` state).

```
On processUnblock.* {
  e.target => ready;
  if ((!empty(running)) &&
      (e.target > running))
  {
    running => ready;
  }
}
```

Verifications and benefits

The immediate benefit of Bossa is that the programmer no longer has to deal with low-level implementation details such as manipulating pointers and lists, which may possibly crash the kernel.

Additionally, several domain-specific properties are enforced by the Bossa compiler. In particular, we are interested in properties that can avoid hazards that may lead the system into a dangerous state.

- Exactly one queue in the `READY` state has to be sorted by the ordering criteria of the policy ; only processes selected from this queue can be given the processor (i.e., moved to the process variable associated with `RUNNING`).
- When assigning a process to a variable, the variable must be empty. This constraint ensures that no process reference is lost due a wrong manipulation.
- Only transitions between process states valid w.r.t. the previously introduced automaton are accepted. By analyzing the manipulation of processes states with respect to the automaton specification, the compiler is able to detect unsafe state transitions. For example, moving a process from `READY` to `BLOCKED` is clearly incorrect.
- There is no event omission in a scheduler specification. By analyzing the specification of a process scheduler, we are able to identify whether the scheduler treats all necessary process events exported/implemented by the OS kernel.

3 Revisiting Bossa as an aspect

When analyzing the Bossa architecture, one can observe that the RTS has been designed to solve a cross-cut problem. In this section we provide evidence that scheduling inherently leads to code tangling with OS

kernels structured into subsystems (e.g. synchronization, drivers). As a result, scheduler implementation could benefit from the use of aspect-oriented techniques.

3.1 Scheduling as an OS aspect

A scheduler depends on a variety of OS kernel mechanisms. Scheduling policies must refer to many different OS subsystems and their implementation is spread over a large part of the OS kernel implementation.

For instance, in priority-based schedulers, priorities may be updated in different parts of the kernel (e.g., when a process blocks waiting for I/O or periodically at a clock tick). Other schedulers may rely on additional information from OS subsystems. For example, in order to avoid starvation in high priority processes, the scheduler can implement a priority inheritance policy by assigning the priority of a high priority process that blocks in the kernel to the process that holds the resource². This requires the scheduler to access the synchronization and file system subsystems.

Another important characteristic of a scheduler is to define in which situations it should preempt the running process so as to select another process. Preemption is usually performed when the running process finishes its execution (normally, killed by another process or due to an exception), voluntarily yields the CPU or is blocked in the kernel. One typical preemption point in a priority-based scheduler is the arrival of a high-priority process in the ready queue. The arrival of such a process can have different causes (e.g., immediately after the creation of a process or when a process becomes runnable due to a timer expiration). In time-sharing schedulers, preemption occurs when a process expires its CPU quantum. These different preemption points are related to different kernel mechanisms (e.g., system timers, process creation and destruction primitives).

The dissemination of scheduling-related code shown by the previous examples provides strong evidence that the scheduler crosscuts the kernel: a scheduling policy thus forms an OS aspect.

The Bossa approach to programming schedulers may seem quite different from AOP. Indeed, AOP is frequently assimilated to macro processing (e.g., “insert a method call at the beginning of every method that returns an `Int`”) or reflection (e.g., “intercept calls to any method of class `Foo` in order to increment its second argument”). Bossa’s approach of decoupling scheduling issues from the OS kernel by means of an event bus does not really fit either technique. In the following we motivate that it can be advanta-

geously integrated in the so-called Event-based model of AOP [5].

3.2 Bossa and Event-based AOP

AOP frameworks should provide aspect languages for the concise definition of aspects and appropriate weaving technology. On the language level, crosscutting is one of the key notions of AOP. Crosscuts denote different program points where aspects modify the execution of the underlying program — most frequently by inserting new functionality. As motivated above, execution points where processes are created, preempted, destroyed or their priorities changed are examples of such points in OS code for scheduling purposes. Furthermore, at these points, information must be transferred between the OS subsystems and the scheduler: e.g., when a process terminates, the scheduler must be called with the identity of the terminating process.

These examples highlight an essential feature for an AOP framework: a mechanism which abstracts programs (code or executions) to points of interests, i.e., points where information is needed from and where new behavior is to be inserted.

Bossa can be integrated smoothly into an AOP framework recently proposed by Douence and Südholt, Event-based AOP [5]. EAOP uses events and relations between events for crosscut definition and (conceptually) relies on execution monitors to weave aspects into the base program. AOP’s framework characteristics are materialized in EAOP as follows:

- The points of interest of a program execution are defined in terms of events emitted during program execution.
- Points of interest are denoted by patterns of events to be matched.
- Once a pattern has been matched, the base program execution is suspended, an action is executed and the base program is resumed.

EAOP is a very general yet operational model for AOP. It offers a natural abstraction in terms of events, enables the explicit definition of complex crosscuts by means of event complex patterns and accommodates very general actions. Moreover, it allows dynamic weaving: an aspect can be plugged (i.e., woven) at run-time.

Bossa can be clearly seen as an instance of this approach to AOP. Bossa’s scheduling-specific events can be interpreted as EAOP events. After emission, Bossa events are stored and processed at some later execution point, which corresponds to event matching in EAOP. Finally, event processing in Bossa consists in executing actions that implement a scheduling policy.

²Note that another strategy is to terminate the low priority process that holds the resource.

4 Toward an AOP-based OS kernel

Conceptually, the implementation of EAOP relies on an event-based infrastructure and a monitor (for action execution). Bossa fits this implementation model since its implementation uses an event bus for event generation/notification and a scheduler module for event processing.

The Bossa DSL and its event model blend quite well with EAOP which can be seen as a systematic framework for expression of scheduling policies. We believe that AOP is more generally promising as an engineering approach for the implementation of scheduling policies.

Indeed, the techniques outlined in this paper could be applied to other kernel subsystems such as networking, disk scheduling and memory management. These subsystems rely on strategies that could be expressed using DSLs and implemented using AOP.

The resulting aspects could then serve as building blocks for a complete AOP-based kernel framework to allow the implementation of both configurable and reliable OSes. This ambitious goal offers many research opportunities. We detail here three of them.

First, each kernel subsystem should be studied in order to design a DSL to program strategies relevant to this subsystem. This work requires to study trade-offs between expressiveness and safety. For instance, Bossa is very expressive and still supports static verification of the correctness of scheduling properties.

Second, it can be tedious to modify kernel subsystems in order to generate events. Systematic means to define and generate events should be provided. Bossa's current implementation is ad hoc: event generation has been inserted manually at the "right" places in the kernel code. A more comprehensive approach to kernel engineering could provide events related to the implementation such as "the method `f○○` is called" or "the variable `bar` is assigned." Such events could be used to define higher-level events such as "the cache becomes invalid" or "a packet has been lost." It is important to note that application domains can impose constraints on the implementation. For example, in the case of scheduling instructions that implement the event bus must be carefully placed in the kernel to avoid synchronization problems.

Third, for the sake of efficiency, optimization should be studied. Partial evaluation is a good candidate. In some case, it could suppress the monitor by inlining event-related code of the monitor in the kernel.

References

- [1] L. P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, Paris, France, March 26–28 2002. To appear.
- [2] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia, Seattle, Washington*, November 1997.
- [3] H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS'99)*, Florence, Italy, June 1999.
- [4] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering conference*, pages 88–98, Vienna, Austria, September 2001.
- [5] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*. Springer Verlag, September 2001.
- [6] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 261–276, December 1999.
- [7] G. Kiczales, J. Lamping, A. Menhdhekar, et al. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.
- [8] J. Lorch and A. Smith. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks*, 3(5):311–324, October 1997.
- [9] G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Méryllon, and L. Réveillère. Towards robust oses for appliances: A new approach based on domain-specific languages. In *ACM SIGOPS European Workshop 2000 (EW'2000)*, October 2000.
- [10] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 184–197, October 1997.
- [11] P. Pillai and K. G. Shin. Real-Time dynamic voltage scaling for Low-Power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 89–102, Banff, Canada, October 21–24 2001.
- [12] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings*

of the 36th ACM/IEEE conference on Design Automation Conference (DAC'99), pages 134–139, New Orleans, USA, June 1999.

- [13] D. A. Solomon. *Inside Windows NT*. Microsoft Press, 1998.
- [14] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE ACM Transactions on Networking*, 5(4):475–488, August 1997.