Bossa: a Language-based Approach for the Design of Real Time Schedulers

Luciano Porto Barreto and Gilles Muller
COMPOSE group, http://compose.labri.fr/
INRIA/LaBRI/Enseirb, 33405 Talence Cedex, France
Luciano.Barreto@labri.fr Gilles.Muller@inria.fr

Abstract

Over the years, OS developers and researchers have been continuously implementing new scheduling policies to address the requirements of real-time applications. Such schedulers are especially designed to ensure timing guarantees that traditional schedulers are unable to provide. However, writing schedulers is a difficult task. It requires understanding the operation of multiple low-level kernel mechanisms. To complicate matters, schedulers are highly optimized components. Such optimizations hinder code maintenance and testing, and make development prone to errors.

In this paper we present the design of Bossa, a language and its run-time system that are specifically targeted toward the development of scheduling policies. The Bossa language provides high-level abstractions that are specific to the domain of scheduling. These abstractions permit the separation of the scheduling policy from the rest of the kernel, easing the development task. Additionally, a Bossa policy can be verified so as to ensure important safety properties.

We illustrate our approach by presenting the Bossa implementation of an EDF process scheduler and a fixed-priority scheduler that controls other schedulers. We also describe the implementation of the Bossa run-time system, which permits schedulers to be loaded into the Linux kernel.

1 Introduction

In real-time systems, the task of the Operating System (OS) is to correctly schedule processes so that application deadlines are always satisfied. Concretely, the developer must map real-time constraints and requirements of the application onto the scheduling mechanisms offered by the OS. On most commercial OSes, these mechanisms consist mostly of offering different levels of process priority. As a result, finding a mapping that accurately expresses application requirements is a difficult task.

On research and recent commercial OSes, more sophisticated scheduling policies such as Earliest Deadline First (EDF) and Rate Monotonic (RM) [13] are seldom offered. Since these policies permit requirements to be expressed in terms of time values, the developer's task of mapping application requirements to such scheduling policies is easier than assigning priorities. Still EDF or RM do not handle well all classes of applications, in particular multimedia ones [26].

As demonstrated by the number of recent studies in the domain of scheduling for real-time systems [1, 4, 12, 11, 18, 27], there are many situations in which an application-dedicated scheduling policy could do a better job than the default policies provided by the OS:

- **Resource-based scheduling**. Several recent studies propose to implement scheduling based on the notion of application progress [6, 24]. In these approaches, a metric defines whether an application is late or executing at a reasonable progress rate. This information is in turn used to allocate more or less CPU time to the process. The definition of a metric often relies on system resources such as the fill level of a queue [24]. Allowing an application developer to define new metrics implies that the scheduler should be easily extensible.

- **Emerging needs**. New application domains are continuously imposing new constraints on schedulers. In the past 10 years, this has been the case for multimedia applications, thus leading to a new class of scheduling policies [3, 5, 7, 16, 26]. Current research now tries to integrate energy requirements into the scheduler so as to increase the battery duration in portable devices [14, 21]. Defining a policy for emerging needs requires performing many experiments so as to validate new ideas. This also suggests the need for an extensible framework for developing scheduling policies.

- **Mixing RT and non-RT applications**. To reduce the cost of hardware platforms, one trend is to embed on a single machine both real-time tasks and more traditional "Unix" services. However, scheduling requirements for these two classes of processes are quite different, and conflicts and interactions between these requirements are not always well understood. As a result, scheduling conflicts (e.g., the priority inversion problem) are solved in an ad-hoc manner in OSes; unexpected scheduling behavior is likely to appear. Solving this problem implies that rules for controlling the processor allocation between application classes must be explicitly expressed. In other words, one should be able to develop schedulers of schedulers.

When the need for a new scheduling policy is recognized, its implementation must be integrated into an existing OS kernel, which raises many problems. Based on an analysis of several RT and non-RT OS kernels such as RT-Linux, Linux and BSD, we have identified the following difficulties in integrating a new scheduling policy:

- **Scheduler development is error-prone**. Schedulers are traditionally written in C or assembly code with intensive usage of macros, type casts and function pointers. This programming style prevents the compiler from catching harmful errors. Because an error in scheduling code is likely to crash the OS, testing a scheduler requires repetitive reboots and deciphering of post-mortem images. Such development practice is tedious and counterproductive.

- **Massive optimizations hinder code maintenance**. Because schedulers are executed very frequently, they should be highly optimized so as to not degrade the 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.

- **Implementation of existing schedulers is spread over the kernel**. Schedulers are 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 [22]. Only few experts are able to fully understand how the scheduler really works, even in well-documented OSes such as Linux and BSD.

- **No recognized framework or tool to ease scheduler development**. While there are well-established extensible frameworks for developing protocols and file system stacks, there is no equivalent for developing schedulers. As a result, development of a new scheduler requires considerable reengineering of existing kernels.

These problems discourage real experimentation on OSes and restrains a wide dissemination of research results. Even in microkernel OSes such as RT-Mach [25], Chorus [19] and QNX [10], it is difficult to change or integrate a new scheduling policy. In industry, the situation is even worse since developers have tight time-to-market constraints. Therefore, developers have little time to devote to a risky task such as implementing a new scheduling policy.

This paper

In this paper we present the design of Bossa, a language dedicated to the development of scheduling policies. The Bossa language provides high-level abstractions that are specific to the domain of scheduling. In particular, it relies on an event model that fully separates the scheduling policy from the rest of the kernel. In a Bossa re-engineered kernel (e.g., Linux or RT-Linux), drivers and other services only signal events. Scheduling decisions are taken by the policy on receipt of such events.

Benefits of Bossa are (i) an easy and safe development; the programmer no longer has to deal with low-level implementation details such as manipulating pointers, which may possibly crash the kernel, (ii) increased robustness; dealing with high-level abstractions makes possible the verification of important safety properties that are specific to the domain of scheduling.

The contributions of this paper are as follows:

- We present the Bossa language by describing its basic abstractions (events, processes, queues) and we detail the implementation of an EDF scheduling policy. This example demonstrates how Bossa improves readability of scheduling policies while retaining program safety and conciseness.

- Second, we describe how a hierarchy of schedulers can be designed in Bossa. Such a hierarchy allows running different classes of applications while clearly expressing the interactions between classes. More precisely, we show how to write Virtual Schedulers, which are schedulers that manage other schedulers (instead of processes). Then we illustrate the design of virtual schedulers by presenting a fixed-priority virtual scheduler.

- Third, we present safety properties that can be verified statically when compiling a Bossa policy. These properties ensure that a policy cannot lead the system into a dangerous state.

- Finally, we describe the Bossa run-time system that allows scheduling policies to be dynamically loaded into the OS kernel. An on-going implementation of the Bossa run-time system within Linux and RT-Linux is presented.

Paper organization

The rest of this paper is organized as follows. Section 2 presents developer needs in programming scheduling policies. Section 3 presents the Bossa language. In Section 4, we assess the benefit of our approach and we describe safety properties ensured by Bossa. In Section 5, we discuss implementation details and performance issues. Finally, Section 6 concludes the paper with on-going work.

2 An analysis of scheduler development

To identify the needs and problems in implementing schedulers, we have conducted an extensive domain analysis based on papers, books and implementations of process schedulers from commercial OSes (e.g., Linux [2], BSD [15], Windows NT [22]), and from the research community [6, 8, 9, 24]. This analysis has allowed us to identify the crucial points in scheduler development in order to determine appropriate language abstractions for Bossa.

As discussed previously, scheduler development should involve two levels: (i) writing process schedulers, (ii) writing schedulers that control other schedulers. In current scheduling infrastructures these two levels are intertwined. Still, our analysis shows that they require slightly different abstractions. Let us examine the different requirements and functionalities of these levels.

2.1 Writing process schedulers

A scheduler must be able to dispatch, suspend, and terminate any process under its supervision. The critical points in the development of a process scheduler involve the definition of the attributes associated with each process, the criteria used to select a process for execution, the circumstances in which the process selection takes place (e.g., scheduling points), and the scheduler interface available to applications.

Managing process attributes and process queues. During the lifetime of a process, the scheduler maintains information about various attributes of the process execution. These attributes include priority, period, computation requirements, execution mode (e.g., kernel or user mode) and deadlines. To specify the precise behavior of a scheduler, it is important not only to define the attributes used for process selection, but also to define how and when those attributes are computed/updated. For example, process priorities are generally recomputed in several parts of the OS (e.g., when the running process blocks or at clock ticks).

Also, a process is associated with a *state*, which represents its actual status of execution (e.g., running, ready to run, blocked, terminated).

A process is usually stored in a queue that holds processes with similar properties (e.g., ready processes or process within a priority range). Some schedulers employ multiple ready queues as in the case of BSD.

Bossa addresses these needs by providing abstractions such as process queues and variables to manage process state transitions, and domain-specific datatypes such as `process` to specify process attributes.

Defining scheduling points. A scheduler is activated in many different places in a kernel so as to update process attributes and check if it needs to give the processor to another process. One typical scheduling 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 be caused by several events (e.g., when a process becomes runnable due to a timer expiration or immediately after being created). In timesharing schedulers, a scheduling point occurs when a process expires its CPU quantum. Identifying scheduling points in existing OSes is difficult because they can be related to different kernel mechanisms (e.g., timers, process creation and destruction primitives). Bossa provides a set of events to define scheduling points and actions to be executed by the policy.

Specifying a process selection criteria. The heart of a process scheduler is the criteria it uses to select a new process for execution. Such criteria are generally related to process attributes and system resource metrics. These attributes can be statically defined (by the user or the system administrator) or dynamically updated by the system. For example, the rate monotonic policy [13] selects the process with the smallest period, which is statically assigned at process creation/admission time. A priority-based scheduler can either use static priorities (common under real-time OSes) or dynamic priorities (in the case of timesharing OSes such as Linux and Windows NT). Other schedulers use deadlines, process latency, slack time or average energy consumption as selection criteria. Bossa addresses these needs with the notion of `ordering_criteria`, which specifies how the scheduler selects a process for execution.

Exporting an interface to applications. A scheduling interface is composed of a set of functions that are used to initialize or update certain system or application attributes used by the scheduler (e.g., priority values, period). Depending on the functionalities exported by this interface, it is possible to adjust/specialize the behavior of a scheduler to particular needs. In other words, the scheduler interface defines the degree of configurability provided by a scheduler. Bossa allows the definition of an interface through which a scheduler exports a set of functions to users.

2.2 Writing schedulers that control other schedulers.

A hierarchical composition of schedulers can provide differentiated service guarantees to applications. Such composition permits the coexistence of multiple scheduling policies that simultaneously provide services to a group of applications. For instance, it is possible to have a best-effort scheduler to handle background and non-critical applications and employ a real-time scheduler (e.g., EDF, rate monotonic) to control processes that have stringent temporal requirements. To control these two schedulers, one may have a fixed-priority virtual scheduler that gives more importance to the real-time scheduler. Let us now analyze the main functionalities of virtual schedulers.

Managing child scheduler states. The state of a scheduler reflects its intention to give control to one of its processes (or child schedulers). A virtual scheduler uses the state of the schedulers it controls in a slightly different way than a process scheduler uses the state of ordinary processes. Consider two schedulers P and Q, where P is a scheduler managed by Q. When an arbitrary process P_i managed by P blocks, it does not necessarily imply that P is *blocked* from the viewpoint of Q. If P has other processes in its ready queue, P must not be considered by Q as a blocked scheduler. P should be treated as blocked only when it has no ready process to schedule. This also applies in the case where P_i unblocks: P may or may not want to reschedule.

Concretely, a process scheduler needs a communication mechanism to notify its parent scheduler about state changes. As for a process scheduler, a virtual scheduler needs to keep track of which schedulers are ready, running, blocked, etc. Bossa addresses the communication among schedulers via a hierarchical event model.

Exporting an interface to other schedulers. Just as a process scheduler exports an interface to its processes, a virtual scheduler has to export an interface to other schedulers. Because a virtual scheduler can potentially handle an arbitrary number of child schedulers with different requirements, it should be able to export several interfaces. Each interface may provide different services simultaneously.

3 The Bossa language

To address the existing needs in scheduler development, we have designed a language to program scheduling policies. Following the two levels identified in our domain analysis, Bossa provides abstractions that permit the development of both Process Schedulers and Virtual Schedulers: process and scheduler states, process attributes and events. Also, since Bossa is strongly typed and has restricted expressiveness we are able to check the safety of a Bossa policy according to important domain-specific properties.

3.1 Writing a process scheduler

A scheduler contains three parts: declarations, event handlers and an interface for interacting with processes managed by the scheduler.

We present how to write a process scheduler by describing code excerpts from the implementation of an Earliest Deadline First (EDF) scheduler. We assume here that the processes managed by the EDF scheduler are periodic (i.e., they have well-known periodic release times). Our EDF scheduler can be easily extended to consider aperiodic and sporadic processes as well.

Process states

A process is associated with a state that describes its current activity. A process can be either in a `RUNNING`, `READY` 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 automaton of Figure 1. When the source state of a transition is statically known, correctness of the transition is checked by the Bossa compiler. Otherwise, the compiler inserts an assertion.

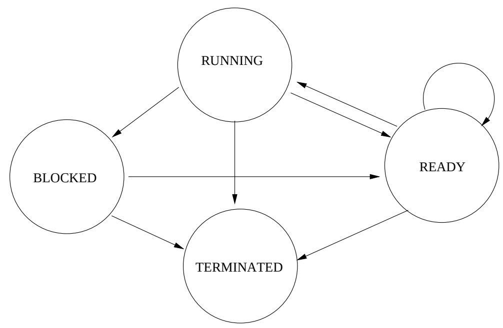

Figure 1: Valid state transitions in Bossa

Scheduler states

A scheduler is a virtual process that splits the CPU time between processes. It also possesses a state: `RUNNING`, `BLOCKED` or `READY`. A scheduler is `RUNNING` when one of its processes runs on the CPU. A scheduler is `BLOCKED` when none of its processes is able to run. Finally, a scheduler is `READY` when one of its processes is ready to run. This is specified by the following pseudo-code (`empty(state)` returns true if there is no process in that state):

```
if (!empty(RUNNING))
  Scheduler_state = RUNNING;
else {
  if (empty(READY))
    Scheduler_state = BLOCKED;
  else
    Scheduler_state = READY;
}
```

The scheduler state is computed after the treatment of an event by the associated policy handler. The code for computing the Scheduler state is generated by the Bossa compiler and can be optimized depending on computations done by the event handler.

Process attributes

Each scheduling policy defines a `process` type that contains attributes to support policy execution. For instance, in EDF, each process has an associated period (`period` attribute) and a worst-case computation time within that period (`wcet` attribute). The `process` type also contains a `deadline` attribute, which specifies the next deadline, and two counters: one to record the missed deadlines (`missedDeadlines`) and another for the CPU ticks used within the current period (`ticks`). This is illustrated by the following Bossa excerpt:

```
process =   {
  time period;
  time wcet;
  time ticks;
  time deadline;
  int missedDeadlines;
};
```

Ordering of processes

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, which contains a list of process attributes and for each attribute a specific order (e.g., lowest, highest).

The EDF policy selects the process with the closest deadline. The excerpt below specifies that the scheduler always selects the process with the smallest `deadline` attribute.

```
ordering_criteria = {lowest deadline};
```

Process queues and variables

Bossa provides process queues and process variables for storing processes. Each queue or variable is 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.

Several domain-specific properties are defined on process queues and variables:

- Since the current version of Bossa is limited to uniprocessors, only a single process variable can be associated with `RUNNING`.
- No process queue or variable can be associated with `TERMINATED`.
- Queues are sorted either by FIFO, LIFO, or the ordering criteria defined in the policy.
- Exactly one queue in the `READY` state has to be sorted by the ordering criteria of the policy (using the attribute `sorted`); only processes selected from this queue can be given the processor (i.e., moved to the process variable associated with `RUNNING`). The function `select()` retrieves the first process of the sorted queue. The compiler enforces the constraint that the queue is not empty.
- Moving a process to a queue or a variable changes the state of the process. Additionally, moving a process to the variable associated with `RUNNING` dispatches it.
- When assigning a process to a variable, the variable must be empty. When accessing a process variable or one of its fields, the variable must be non empty. These constraints are enforced by the Bossa compiler.

Process queues and variables are defined within a `states` declaration. The code excerpt below declares a process variable (`running`) and queues for the EDF scheduler. There is one queue for storing `READY` processes and two `BLOCKED` queues for `yield` and `blocked` processes.

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

Other declarations

Finally, Bossa allows the definition of global variables and constants. For instance, in the EDF scheduler, we declare a constant that corresponds to the maximum usage of the CPU, a `TICK` constant for the value of a tick, a counter for keeping track of worst-case execution time violations, and a global variable that stores the currently used proportion of the CPU.

```
const int MAX_UTIL = 100;
const time TICK = 10; // in ms
int wcetViolations = 0;
int totalUtil = 0;
```

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. Events are generated by kernel scheduling points, which send events to the Bossa run-time system. Table 1 summarizes the events available in Bossa and the corresponding entities that generate them.

Table 1: Bossa events

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

The `processBlock` and `processUnblock` events are structured in a hierarchy (denoted by the `*` symbol). Such an event hierarchy permits the specification of both generic handlers (`processBlock.*`) as well as specific handlers for IO subsystems, synchronization mechanisms and time service.

In Bossa, handlers are parameterized by an event object. The `event` datatype is a structure that contains the event type (`type`), the event creation time (`time`), the process that generated the event (`source`) and the process for which the event was generated (`target`). All event handlers are defined in a `handler` block as illustrated below.

```
handler (event e) {
... code of event handlers ...
}
```

We now describe the main event handlers used to implement an EDF scheduling policy in Bossa: `processBlock.*`, `processYield`, `processUnblock.*`, `processUnblock.Timer`, `clockTick` and `Schedule`.

When a process blocks, the policy moves the process to the `blocked` state and its associated queue using the `=>` operator. This handler treats the complete hierarchy of blocking events.

```
On processBlock.*  {
  running => blocked;
}
```

When a process yields the cpu, it means the process has finished its computation within its deadline. In this case, the policy moves the process to the `yield` queue. The `yield` and `blocked` queues both refine the `BLOCKED` state but carry different semantics. Using multiple queues for storing processes eases policy design by avoiding tedious case testing.

```
On processYield {
  running => yield;
}
```

When the scheduler receives an unblock notification, it moves the process for which the event was generated to the `ready` state. If the process that unblocks has an earlier deadline 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;
  }
}
```

The EDF policy uses a timer to keep track of deadline violations. This timer is initially set when a process is admitted by the scheduler and is reset for the next period whenever the timer expires. At that time, a `processUnblock.Timer` event is generated by the scheduling point in the timer service. It should be noted that the generic `processUnblock.*` handler is not executed for this event, since a specific handler is defined in the EDF policy as presented below.

```
On processUnblock.Timer {
 // denotes the begin of a new period
 e.target.deadline = now() + e.target.period;
 // check the actual process state
 switch e.target in  {
   case yield:
   { // Ok, didn't miss a deadline
     e.target => ready;
     // Preempt the running process ?
    if ((!empty(running)) &&
        (e.target > running))
     running => ready;
   }
   case running:
   { // missed a deadline while running
     e.target.missedDeadlines++;
     globalMissedDeadlines++;
     // preempt the running process
     // obs:e.target = running
     e.target => ready;
   }
   case blocked:
   { // missed a deadline while blocked
     e.target.missedDeadlines++;
     globalMissedDeadlines++;
   }
   case ready:
   { // missed a deadline while ready
     e.target.missedDeadlines++;
     globalMissedDeadlines++;
   }
 }
}
```

Every `clockTick` the scheduler increments the `ticks` counter for the running process. Then the policy checks if the process has exceeded its agreed CPU time. If so, the policy increments the global wcet violation counter and moves the process to `yield`.

```
  On clockTick {
     running.ticks += TICK;
     if (running.ticks > running.wcet) {
       wcetViolations++;
       running => yield;
     }
  }
```

Multiple events may occur nearly simultaneously due to interrupts. After all events have been treated, if the scheduler is `READY` then the run-time system generates a schedule event. To treat a `schedule` event, a policy must select a `READY` process and move it to the variable associated with the `RUNNING` state. The EDF policy handles `schedule` in a simple manner; it uses `select()` to choose the process that has the greatest priority according to the ordering criteria of the policy and moves this process to `running`.

```
On schedule {
  select() => running;
}
```

Scheduler interface

The scheduler interface is a set of procedures that permit application processes to interact with the scheduler. An interface can be used to change process attributes, and to retrieve and set global scheduler parameters.

The EDF scheduler interface consists of a single `register` function that is called to add a process to the scheduler. This function performs admission control. The EDF policy maintains a global variable named `totalUtil` (initially zero), which represents the CPU utilization. When `register` is called to add a new process, the scheduler computes the CPU utilization of the new process by dividing the requested CPU time (`p_wcet`) by the period (`p_period`). Then, the scheduler checks whether accepting this surpasses `MAX_UTIL`. If so, the scheduler refuses the process. Otherwise, the process is accepted and `totalUtil` is updated.

```
interface = {
  RegReply register
  (process p, time p_period, time p_wcet) {
    int util = p_wcet / p_period;
    // admission control
    if (totalUtil + util > MAX_UTIL) {
      return REGISTER_ERROR;
    }
    else { // process is accepted
      totalUtil = totalUtil + util;
      p.ticks = 0;
      p.missedDeadlines = 0;
```

```
      p.wcet = p_wcet;
      p.period = p_period;
      p.deadline = now() + p_period;
      p => ready;
      return REGISTER_OK;
    }
  }
}
```

3.2 Writing a virtual scheduler

Writing a virtual scheduler is similar to writing a process scheduler. A virtual scheduler is associated with a state, it possesses attributes and an ordering criteria to select and compare child schedulers. event treatment, the virtual scheduler state is computed following the same algorithm defined for a process scheduler.

The main difference between a process scheduler and a virtual scheduler is the handling of events. The Bossa run-time system sends the event to the first scheduler in the hierarchy. After receiving the event, a virtual scheduler forwards the event to the appropriate child scheduler and then updates the child scheduler state according to the result of the event treatment.

In some cases, immediately forwarding an event is useless because the scheduler is not running or will not become running due to the event. In such situations, the treatment of the event can be postponed until the scheduler becomes ready. Deferring events improve the response time of complex scheduling hierarchies by making scheduling decisions faster.

Bossa provides two primitives for event forwarding: `forwardImmediate()` and `deferEvent()`. An immediate event is instantly executed by the child scheduler. A deferred event is stored into an event queue associated with the child scheduler and is processed only when the child scheduler is invoked by a later immediate event. In this case, the immediate event is processed after all deferred events. Deferred events are treated by the scheduler in the order in which they were stored in the scheduler event queue.

A fixed-priority virtual scheduler

We now present the implementation of a virtual scheduler that uses a fixed-priority policy to share the CPU among its child schedulers.

The following excerpt declares one scheduler attribute and the ordering criteria.

```
virtual_scheduler Fixed_priority = {
  scheduler = {
    int importance;
  }
  ordering_criteria = {highest importance};
}
```

We then define the queues used to store the schedulers, and a scheduler variable as a reference to the running scheduler.

```
  states = {
    RUNNING running : scheduler;
    READY ready     : queue;
    BLOCKED blocked : queue;
  }
```

For virtual schedulers, handlers are parameterized by an event object and the child scheduler (e.g., `next`) to which the event is destinated. Event handlers are defined in an `handler` block as illustrated below.

```
  handler (event e, scheduler next) {
  ... code of event handlers ...
  }
```

To illustrate how a virtual scheduler forwards events and manages the state of its child schedulers, we now describe the implementation of the `processUnblock` event handler. Our fixed priority virtual scheduler defers all unblocking events that are targeted to a low-priority child scheduler. Such an unblocking event can be executed later, because we know that its execution will not make the virtual scheduler preempt the running child scheduler.

If the child scheduler (`next`) has a greater priority than the running one, the running scheduler is preempted by moving it to `ready`, `next` becomes ready due to the `processUnblock` and the event is stored for being treated when `next` becomes `running`. Note that `next` is not moved to `running` because several events can be generated by the run-time system at a same time. Selection of the new `running` scheduler is done in the schedule event that is called by the run-time system when all events have been processed.

```
   On processUnblock.* {
     if (!empty(running) &&
```

```
      (next > running))
    { // Preempt running
      running => ready;
      next => ready;
      deferEvent();
    }
    else
      if (next in running)
      { // next is running
        running => forwardImmediate();
      }
      else
      {
       deferEvent();
      }
  }

  On Schedule {
    select() => running;
    running => forwardImmediate();
  }
```

The fixed-priority scheduler provides the `register` function, and a function `setImportance` to change the importance attribute of a child scheduler, as illustrated by the following excerpt.

```
 interface = {
   RegReply register (scheduler s,
                      int priority) {
     s.importance = priority;
     s => ready;
     return REGISTER_OK;
   }
   void setImportance (int priority) {
     s.importance = priority;
   }
}
```

4 Assessment

We now discusss how Bossa eases the development of scheduling policies and enforces the verification of safety properties.

Ease of development

The EDF scheduler shows how the high-level abstractions provided by Bossa make it easier to program a scheduler policy. There is no need for dealing with pointers or manipulating other complex data structures. As a result, the developer can concentrate on writing the policy code instead of dealing with low-level implementation details.

Virtual scheduler development is a natural extension to the implementation of process schedulers since both basically rely on similar abstractions (e.g., states and transitions, queues, event handlers). Such common level of abstraction eases the experimentation for process developers with hierarchical schedulers. The proposed extensions for hierarchical event handling clearly define the interaction between schedulers, which facilitates the understanding of how scheduling decisions are performed by the schedulers in the hierarchy. These extensions also permit a straightforward integration of a process scheduler as a black-box component in an scheduling hierarchy (i.e., there is no need for code changes).

Properties of the Bossa language

A major advantage of using Bossa is that it eases the verification of scheduler policies. In particular, we are interested in properties that can avoid hazards that may lead the system to a dangerous state. We now discuss these properties in more detail.

Only ready processes are eligible for execution. We guarantee that only processes that are in the ready state can be dispatched by a process scheduler. This property prevents a scheduler from giving control to a process, such as a blocked process, that is incapable of executing. To do so, we check that the ordering criteria must only refer to processes in the `READY` state. This restriction prevents the scheduler from dispatching a blocked or a running process.

No loss of process references. In Bossa, we restrict process manipulation such that a process can only be moved between two valid states. This restriction ensures that a process can not be present in two or more process states simultaneously or left by mistake in an undefined state. This property avoids inconsistent situations such as a process being both in a ready queue (i.e., runnable) and in a blocked queue waiting for a system event to occur.

Only valid process state transitions are accepted. Bossa defines a set of valid state transitions involving processes (or schedulers) according to the state transition automaton shown in Figure 1. By analyzing the manipulation of processes states with respect to the automaton specification, we are able to detect unsafe state transitions. For example, moving a process from ready to a blocked state 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. For example, a scheduler that ignores a process termination event seriously compromises the OS operation because it will not immediately dispatch another process for execution, which may delay the overall system execution or cause the system to halt.

Isolation among schedulers is guaranteed. In a hierarchical scheduler model, it is essential to guarantee that a scheduler cannot interfere with the operation of another scheduler. A scheduler in the hierarchy should only be able to preempt a process that belongs to one of its child schedulers. In addition, a scheduler cannot dispatch a process in a processor held by another scheduler.

Scheduler execution is bounded. Bossa provides neither unbounded loops nor recursive functions. These language restrictions allows us to bound the execution time of a scheduling policy.

5 Implementation

We are currently implementing Bossa within Linux and RT-Linux. The main task is to re-engineer the kernel so as to separate the scheduler from other kernel subsystems. For instance, scheduling points in the OS are modified to generate events to the Bossa run-time system.

The Bossa compiler translates a scheduling policy into C code that is then compiled as a Linux kernel module. At load time, the kernel module registers with the scheduler hierarchy and starts treating the event notifications sent by the kernel. Each scheduler module has a specific device entry in the `/dev/` hierarchy. This allows the scheduler interface to be accessible through conventional `ioctl` operations. Additionally, the resulting kernel module can embed functions that provide statistics about its operation through the `/proc` file system.

For performance purposes, the C code generated by the Bossa compiler is linked with a target-specific library that implements queues and events. Thus, for each architecture or system, it is possible to provide specific optimizations. As a result, scheduler developers do not have to manually code low-level optimizations.

6 Conclusion

In this paper we have described Bossa, a language framework for the development of process scheduling infrastructures. As showed by our examples, Bossa is able to express a wide range of scheduling policies while providing safety properties. We are currently developing an initial Bossa prototype for both Linux and RT-Linux. We are also refining the language and evaluating the performance of the framework by analyzing its effects on real applications.

Availability

Examples of scheduling policies and a first version of Bossa are available at:
`http://compose.labri.fr/prototypes/bossa/` .

Acknowledgments

The authors would like to thank Julia Lawall and the members of the Compose group for their helpful comments. This work is part of the RNRT Phenix project and is supported by France Telecom under the CTI contract number 991B726.

References

[1] A. Atlas and A. Bestavros. Design and implementation of statistical rate monotonic scheduling in KURT Linux. In *IEEE Real-Time Systems Symposium*, pages 272–276, 1999.

[2] M. Beck, H. Bohme, M. Dzladzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, Reading, MA, USA, 1998.

[3] 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.

[4] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportional fair scheduling in multiprocessor servers arbitrary deadlines. In RTAS'2001 [20].

[5] 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.

[6] J. Douceur and W. Bolosky. Progress-based regulation of low-importance processes. In SOSP'99 [23], pages 247–258.

[7] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In SOSP'99 [23], pages 261–276.

[8] B. Ford and S. Susarla. CPU Inheritance Scheduling. In OSDI'96 [17], pages 91–105.

[9] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In OSDI'96 [17], pages 107–121.

[10] D. Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, USA, April 1992.

[11] K. Jeffay and G. Lamastra. A comparative study of the realization of rate-based computing services in general purpose operating systems. In *Proceedings of the Seventh IEEE International Conference on Real-Time Computing Systems and Applications*, pages 81–90, December 2000.

[12] K. Jeffay, F. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real-Time Systems Symposium*, December 1998.

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

[14] J. Lorch and A. Smith. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks*, 3(5):311–324, October 1997.

[15] M. McKusick, K. Bostic, and M. Karels. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, May 1996.

[16] 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.

[17] *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, Berkeley, October 1996.

[18] J. Regehr and J. A. Stankovic. Augmented CPU reservations: towards predictable execution on general-purpose operating systems. In RTAS'2001 [20].

[19] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, USA, April 1992.

[20] *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'2001)*, Taipei, Taiwan, May 2001.

[21] 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.

[22] D. A. Solomon. *Inside Windows NT*. Microsoft Press, 1998.

[23] *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, December 1999.

[24] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)*, pages 145–158, February 1999.

[25] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In *USENIX Mach Workshop Conference*, pages 73–82, Berkeley, CA, USA, October 1990.

[26] 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.

[27] W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In RTAS'2001 [20].