# How to Rapidly Prototype a Real-Time Scheduler

Luca Abeni
ReTiS Lab
Scuola Superiore S. Anna, Pisa, Italy
luca@sssup.it

John Regehr
School of Computing
University of Utah
regehr@cs.utah.edu

## Abstract

*Implementing a new scheduling algorithm in an OS kernel is often an important step in scheduling research because it permits evaluation of the algorithm's performance on real workloads. However, developing a new scheduler is not a trivial task because it requires sophisticated programming skills and a deep knowledge of kernel internals. In this paper we show how to use the HLS scheduling framework to develop new schedulers in a user-level simulator, where advanced debugging tools can be used to achieve a high level of robustness before the scheduler is converted to a loadable kernel module simply by recompiling it. Besides facilitating debugging and porting, the HLS abstraction has the benefit of bringing the programming model very close to what, in our experience, scheduler developers want.*

## 1. Introduction

In recent years there has been a lot of interest in integrating real-time techniques in general-purpose operating systems (OSs) to run, for example, time-sensitive applications in desktop environments. New and interesting real-time scheduling algorithms have been developed to meet the requirements of these applications.

One of the most critical aspects of this kind of research is to implement a proposed algorithm in a real kernel so it can be validated and evaluated. However, such an implementation requires a deep knowledge of the kernel and better than average programming skill. Systems software programming is more complex than application-level programming because there are many non-obvious rules about what kinds of actions can be performed in different parts of the code, because systems code is often highly concurrent, and because some important tools, such as source-level debuggers, are not usable in the kernel. Although most kernels provide debugging support, these kernel debuggers are usually much less sophisticated than comparable user-space tools. Moreover, the implementation of the new scheduling algorithm is often tightly bound to a specific kernel organi-

zation — this can be a serious problem with continuously evolving systems like Linux.

Some research projects, like MaRTE OS [8] and Shark [2], have proposed solutions for implementing user-defined scheduling policies, but the implemented schedulers have not been shown to be portable and they cannot be used in common general purpose OSs such as Windows or Linux. Bossa [4], on the other hand, supports loadable schedulers in the Linux kernel using a safe domain-specific language, eliminating the possibility of crashing the kernel with a stray pointer and also enabling high-level checks on the protocol that exists between schedulers and the kernel. Functional testing, however, must still be performed in the difficult kernel environment.

In this paper, we propose a solution for these problems where schedulers are portable and can be debugged entirely at user level. Our solution is based on the Hierarchical Loadable Schedulers (HLS) framework, an infrastructure originally developed for hierarchically composing soft real-time schedulers [7]. Here we exploit another useful property of HLS: its ability to separate a library of HLS schedulers from several OS-dependent *backends*. This is accomplished through a clean separation between portable scheduling *policies*, implemented by loadable schedulers, and non-portable scheduling *mechanisms*, implemented in the HLS backends. The *HLS application programming interface* (HLS API) provides a common language for dialogue between schedulers, and between schedulers and backends. A real-time researcher or a student can rapidly develop a portable scheduler in user space, testing it using simulated task sets, and then, once it is debugged, insert it into a supported kernel. We currently support Linux and Windows 2000.

## 2. The HLS Framework

Since HLS framework was developed for hierarchically composing schedulers there are child/parent relationships between schedulers. Each scheduler only communicates with its parent scheduler, its child schedulers (note that there

can be more than one), and the HLS infrastructure. All these interactions happen according to a well-defined interface: the HLS API. In this paper we give only a quick overview of HLS concepts.

A child scheduler can **register** and **unregister** itself with a parent, **request** allocation of a physical CPU from its parent, **release** a CPU that it is using (block), and **send** messages to the parent (e.g. to request a change in priority, share, or deadline). On the other hand, the parent scheduler can **grant** a CPU to a child scheduler, and **revoke** a CPU from the child (i.e. preempt it). Finally, interactions between the infrastructure and a scheduler happen when the scheduler is initialized/deinitialized or when a timer requested by the scheduler fires.

An important invariant provided by the HLS API is that each scheduler is aware, at all times, of the number of physical processors allocated to it. In other words, the HLS API is in the style of *scheduler activations* [1].
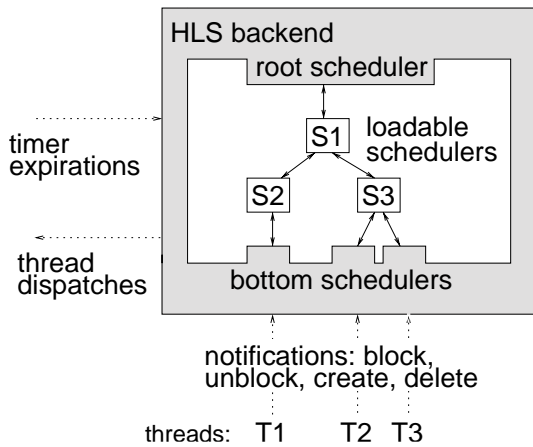


**Figure 1. The structure of HLS: the shaded backend isolates portable schedulers from the rest of the system**

Schedulers are not permitted to interact except through the HLS API. All communication between schedulers is synchronous and is efficiently implemented through direct function calls. Finally, there is no concurrency within the scheduling hierarchy: all HLS entry points are explicitly serialized using a spinlock.

To provide a uniform interface, each system processor is represented as an HLS *root* scheduler that has some special properties: every root scheduler has exactly one child and it never revokes the CPU from the child. Similarly, every thread in the system is represented as a special `bottom` scheduler that explicitly gives up the processor each time the thread blocks, and requests the processor each time it unblocks. Root and bottom schedulers are "spe-

cial": they provide glue that is necessary for the HLS backend to present the standard HLS API to all loadable schedulers. Because loadable schedulers interact with each other, and with the HLS backend, through the standard HLS API, schedulers are agnostic with respect to their position in the scheduling hierarchy. This structure is depicted in Figure 1.

When developing a new scheduler it is possible to implement only a subset of the full HLS interface. Two restrictions that make development significantly easier are uniprocessor schedulers that control the allocation of at most one physical processor, and *root-only* schedulers that do not handle processor revocation. For example, it took one of us less than a day to implement a root-only, uniprocessor proportional share scheduler.

## 3. HLS Backends

A backend for HLS that runs in the Windows 2000 kernel was described in [7]. In this section we describe two additional HLS backends. The first implements an event-driven simulation that facilitates testing of HLS schedulers in a user-level process and the second supports HLS schedulers in the Linux kernel.

### 3.1. Simulation

The simulator backend accurately simulates the behavior of a kernel exporting the HLS API. The simulated task set is described in terms of number of tasks, duration of each task, distribution of the execution time of each task instance (i.e., distribution of the time for which a task executes before blocking), distribution of the interarrival time of each task, and so on. Although the simulator is always deterministic, nondeterminism can be emulated by seeding the pseudo-random number generator with a true random number.

Special events can be defined that change task scheduling parameters and move tasks from one HLS scheduler to another. For example, a user who starts up some multimedia applications can be modeled by defining an event that moves tasks from the default time sharing scheduler to a real-time scheduler.

Although loadable schedulers in a real OS kernel are dynamically loaded at run time, we currently statically link schedulers into the simulator. Scheduler instances are then dynamically created when the simulator starts up, and therefore the scheduling hierarchy is defined by some C code in the simulator. We are considering making the hierarchy configurable at execution time or describing it in a text file parsed by the simulator at startup.

The output of the simulator is a trace describing the generated schedule, which can be parsed using some filters (that are distributed with HLS) to generate graphical representations of task executions.

### 3.2. Linux

We implemented HLS on Linux as a kernel module that must be loaded into the kernel before any scheduler modules can be loaded. The HLS module interfaces with the Linux kernel by intercepting scheduling events (described in Section 2), and exports the HLS interface to loadable schedulers.

To intercept the proper events the HLS module must be inserted in a patched Linux kernel that exports 6 hooks: `fork_hook`, `cleanup_hook`, `block_hook`, `unblock_hook`, `setsched_hook`, and `getsched_hook`. The patch is not large or complicated: it changes only a few files in the Linux kernel, adding a total of less than 150 lines of C. For every hook the patched kernel contains a function pointer that is initially set to null. When the HLS module is inserted it sets these pointers to its handlers, so that the kernel will invoke the proper HLS handlers when processes are created, destroyed, blocked, and so on. The HLS handlers in turn convert each one of these events into an HLS event, invoking the proper HLS scheduler according to the HLS API.

When a scheduler is inserted into the kernel it registers itself with the HLS infrastructure. New instances of this scheduler can be created using the Linux proc filesystem. A scheduler instance can be set as default scheduler, so that all threads will be scheduled by it.[1] The Linux backend makes extensive use of the proc filesystem for exporting the scheduler hierarchy to the user and for visualizing information about the registered HLS schedulers and currently existing scheduler instances. When the HLS infrastructure is loaded, some new procfs entries are created, and when a scheduler is loaded or a new instance is created they are updated.

Threads can be moved between scheduler instances using the `sched_setscheduler()` syscall. To make this possible we extended `sched_setscheduler()` in a backward compatible way: if the syscall is invoked using the standard syntax, it will behave as usual, but new "HLS aware" programs can use some extended fields in the `sched_param` structure to access the new schedulers. Of course, if this extended `sched_setscheduler()` syntax is used on a standard kernel, or when the HLS module is not loaded, the call will fail and return an error code.

Implementing a new scheduler in a general-purpose OS always begs the question of whether the basic OS mechanisms are capable of supporting real-time applications. We believe that this problem should be considered separately from the problem of implementing new scheduling algorithms. There are well-known solutions [3] to problems such as dispatch latency and coarse-grained timers. These solutions are almost completely orthogonal to HLS and we

---

[1]When the default scheduler is created, all existing threads are moved to it and when a new thread is created it is assigned to the default scheduler.

assume that the HLS implementation can make use of them when they are available.

## 4. Developing a New Scheduler

The implementation of a new scheduling algorithm usually goes through three stages (in practice, of course, there may be overlap between these stages).

In the **first stage**, the scheduling algorithm is fleshed out: the developer's main goal at this point is to create code that does not crash the system. That is, it must respect all relevant operating system invariants (e.g. don't block in certain kernel contexts, permit only one running thread per processor) and avoid corrupting memory (e.g. by dereferencing a null pointer or walking off the end of an array).

HLS helps the developer avoid breaking some invariants because its restricted API simply does not provide functions that cannot be safely called from the scheduler. The HLS simulator is also helpful during this stage because it can subject the scheduler to systematic deterministic and randomized testing to elicit errors. User-mode testing is preferable to in-kernel testing for many reasons. First, advanced debugging tools such as Purify, Valgrind, ElectricFence, and gdb/ddd can be used on the scheduler's code. Second, when a bug is found it crashes only a process and not a real machine — system crashes are slow to recover from, they typically destroy useful machine state that would have helped debug the problem, and sometimes they cause lasting filesystem corruption.

Finally, HLS takes advantage of the fact that the simulator can invoke a scheduler both much more frequently and much less frequently than a real system would. For example, while a real scheduler is usually called at most a few hundred times per second, the simulator can invoke it hundreds of thousands of times per second.

The **second stage** of the scheduler's development is functional testing. Once a scheduler can handle arbitrary workloads without crashing, the developer's next concern is ensuring that it actually implements the specified scheduling algorithm in all cases. In our experience this is quite easy for simple algorithms like priority and EDF, but can be more difficult for schedulers that tend to have significant internal state, such as those providing CPU reservations [5]. Also, details like dealing correctly with missed timer interrupts due to unfriendly kernel code can be difficult to get right.

The HLS simulator is helpful in this stage of development because it provides an idealized environment where simple and complex workloads can be provided to a scheduler and the results quickly analyzed, e.g. with a Perl script or a graphing program. Perhaps the most important property of the simulator at this stage is determinism — troublesome cases can be replayed over and over until the correct

solution is found. It is also possible to simulate hardware configurations not available to the developer, such as a 5-processor machine.

In the **third stage** of the scheduler's development, it is tested on real applications to show real-world benefit when compared to other schedulers. This typically motivates a number of minor changes to the scheduler, for example to tweak constants and speed up performance-critical code. Since the HLS simulator does not yet model operating system overheads or the detailed behavior of real applications, at this point the developer should migrate her scheduling code into a real kernel in order to run real benchmarks and application code. Fortunately, this can be accomplished through a simple recompilation against a different set of header files and libraries.

HLS provides two main benefits at this stage. First, since the scheduler is implemented as a loadable kernel module, a buggy scheduler can be unloaded and a new version loaded without rebooting the machine, assuming that the bug did not corrupt kernel memory. And second, since a loadable scheduler augments, rather than replacing, the default time-sharing scheduler, it is possible to schedule as few or as many threads under the new scheduler as is desired.

At this point it also makes sense to use a scheduling analysis tool to ensure that the schedule produced by inserting the scheduler in a real kernel matches with the schedule produced by the simulator. For example, we used Hourglass [6] to verify the consistency between the results obtained with the Linux backend and the simulated results.

## 5. Status and Availability

The Windows 2000 backend, described in [7], is complete and functional but cannot be freely redistributed. The Linux and simulator backends are free software, released under the GNU Public License, and can be downloaded here:

```
http://feanor.sssup.it/~luca/hls
```

This package includes a proportional share scheduler, a CPU reservation scheduler, and a priority/round-robin scheduler.

The Linux backend is still under development, but it works on version 2.4 kernels on uniprocessor x86 and PowerPC systems. Some infrastructure for supporting SMP systems is already in place, but we currently do not support SMP on Linux. We are currently debugging a backend that works in the Linux 2.5 kernel. The simulator backend supports multiprocessors, but still needs more work in the area of workload definition and the extraction of statistics about task execution.

## 6. Conclusions

We have presented a novel method for rapidly prototyping new real-time schedulers based on Hierarchical Loadable Schedulers (HLS). HLS has numerous advantages over the traditional method of ad-hoc changes to OS internals. For common classes of scheduling algorithms a good programmer can expect to write a new scheduler that runs in a real OS kernel in a few days or less. We have also described a new backend for HLS that is freely redistributable and runs in the Linux kernel.

## References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 95–109, Pacific Grove, CA, October 1991.

[2] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[3] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on general-purpose operating systems. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[4] Julia L. Lawall, Gilles Muller, and Luciano Porto Barreto. Capturing OS expertise in a modular type system: The Bossa experience. In *Proc. of the ACM SIGOPS European Workshop*, Saint-Emillion, France, September 2002.

[5] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. In *Proc. of the IEEE Intl. Conf. on Multimedia Computing and Systems*, May 1994.

[6] John Regehr. Inferring scheduling behavior with Hourglass. In *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pages 143–156, Monterey, CA, June 2002.

[7] John Regehr and John A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. of the 22nd IEEE Real-Time Systems Symp.*, pages 3–14, London, UK, December 2001.

[8] Mario Aldea Rivas and Michael Gonzalez Harbour. Posix-compatible application-defined scheduling in MaRTE OS. In *Proceedings of the 14th IEEE Euromicro Conference on Real-Time Systems*, Wien, Austria, June 2002.