Proceedings of the International Workshop on Advanced Operating Systems Services (ARTOSS 2003), Porto, Portugal, July 1, 2003.

Mutual exclusion in operating systems with application defined scheduling

Paolo Gai
RETIS Lab
Scuola Superiore S. Anna, Pisa
pj@sssup.it

Giorgio Buttazzo
Dept. of Computer Science
University of Pavia (Italy)
buttazzo@unipv.it

Abstract

This paper presents the shadow mechanism*, a mechanism that enables independence between the implementation of scheduling algorithms and mutual exclusion protocols. The mechanism can be used in operating systems that support the definition of application-defined schedulers using implementation-independent interfaces.*

The proposed method is simple to implement and introduces a negligible run-time overhead, enabling the implementation of mutual exclusion protocols, like Priority Inheritance, Priority Ceiling, and Stack Resource Policy, in a way that is independent of the scheduler structure.

The paper discusses advantages and shortcomings of the approach, illustrating sample code taken from a real implementation done in the S.Ha.R.K. Kernel.

1 Introduction

The increasing power offered by modern computer systems, together with the increasing performance required by new applications, create an additional demand for flexible operating systems, which should be easily portable to different platforms and adaptable to different application requirements.

New techniques have been recently developed to allow a modular specification of scheduling and synchronization mechanism, enabling applications to the usage of proprietary application-defined scheduling algorithms. The proposed models are different, ranging from the modification of the operating system internals to generic interfaces for middleware user-level scheduling.

These models, like the one adopted in S.Ha.R.K. [4] and in MarteOS [7], propose an interface that is suitable for shared memory systems (some of them enhances the POSIX Pthread library). Communication among tasks is performed by accessing shared buffers, and tasks that concurrently access the same shared resource must be synchronized through *mutual exclusion*. In these cases, real-time theory [8] teaches that mutual exclusion through semaphores is prone to *priority inversion*. In order to avoid or limit priority inversion, suitable resource access protocols have to be used, like for example Priority Inheritance and Priority Ceiling (note that these protocols have been accepted into the standard interfaces for operating systems, proposed by the POSIX 1003.1b and OSEK standards).

All the new interfaces that propose an application defined scheduling provide a coherent redefinition of the behavior for the mutual exclusion, which require at least the support for the same functionality provided in these standards. The current approach in this redefinition is to provide access to a set of events/function calls that cover the behavior of both the scheduling part and the mutual exclusion part.

When defining an interface for application-defined schedulers, an important issue is to avoid duplication of code, because users would like to write things once, and then reuse them everywhere. The definition of application-defined schedulers that are too local would prevent the schedulers to share an application-defined mutual exclusion protocol implementation. Also, the tasks scheduled by different schedulers could not share application-defined mutexes as well, because each scheduler could not access the internal data representation of other schedulers.

Moreover, for the same reason, local application-defined policies could not support application-defined mutual exclusion of system-wide resources that have to be shared among tasks (for example, when accessing hardware devices).

Users usually think of scheduling algorithms and mutual exclusion protocols as two separate mechanisms. Whereas a lot of scheduling algorithms have been proposed in the real-time literature, the number of mutual exclusion protocols is quite limited. If only one interface is provided for specifying both scheduling and mutual exclusion, the user is forced to implement a mutual exclusion protocol every time a new

scheduling algorithm is included in the kernel.

We believe that a good kernel interface should allow the specification of a scheduling algorithm in a way that is independent from the specification of the mutual exclusion protocol. This approach would allow the following advantages:

- treating a scheduling algorithm in separation from the mutual exclusion protocol;
- implementing an application-defined scheduler without caring about the mutual exclusion protocol;
- implementing a synchronization mechanism that can be applied to tasks belonging to different application-defined schedulers, providing both local and system wide application-defined mutual exclusion protocols, independent of the implementation of a single algorithm.

This paper proposes a mutual exclusion synchronization method, called the *shadow mechanism*, which is independent from the scheduling algorithm. Advantages and shortcomings of the proposed approach are analyzed throughout the paper.

The shadow mechanism, that has been implemented on the S.Ha.R.K. operating system [4], enables resource sharing between tasks in a way that is independent of the application-defined scheduler. The method can be implemented with little effort and with negligible run-time performance. A simple implementation independent interface can be easily derived from it.

One of the main advantages of this method is its *independence* from the implementation of the scheduling algorithms. Once a synchronization protocol is written and tested *once*, it can be reused *for free* in all the application schedulers that will be written in the future.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents the overall architecture of the shadow mechanism. Section 4 illustrates how the mechanism has been implemented in S.Ha.R.K. Section 5 briefly presents what is needed on a generic system for implementing the mechanism together with an implementation independent interface, and finally Section 6 states our conclusions and future work.

2 Related works

In the last few years, new approaches have been described for developing application-defined scheduling. For example, in the MarteOS project [7], a novel scheduling framework has been proposed consisting of scheduler threads that communicate with the kernel using an event passing mechanism. This approach uses application defined mutexes and a set of events to address the problem of synchronization, when the user wants to specify its own mutual exclusion policy. The user needs to use the global POSIX mutexes when mutual exclusion needs to be implemented among tasks of different applications.

Other systems, instead of using an event based approach, preferred to use a set of function calls inserted in the kernel code to abstract from a particular scheduler implementation. Examples of this approach are the S.Ha.R.K. Kernel [4], and the upcoming modular scheduler of FreeBSD [9]. In the following, we will use the term *callback* to refer to a function call or an event (depending on the implementation) that implements a particular behavior in the scheduling algorithm.

An approach similar to the one proposed in this paper has been implemented in the context of the CPU Inheritance Scheduling [3], which provides some kernel mechanisms to "inherit" CPU time from one task to another. In this way, the kernel provides only the basic mechanisms used by application tasks to implement the scheduler: using this approach a blocked task "donates" its execution time to the blocking task waiting for the release of the blocking resource.

Another approach that is worth citing is the Bandwidth Inheritance Algorithm (BWI) [5], that differs from the method proposed in this paper because inheritance is done in a way that the inherited task consumes the bandwidth of the blocked task [1]. Please note that the approach cannot be applied in general, because it may be that two application-defined schedulers do not share information about available bandwidth, priorities and deadlines that are needed to implement the BWI approach.

Aiming at a proposal for standardization, our approach is to follow a generic method to implement synchronization that is independent of the implementation of each application-defined scheduler: only with the orthogonality between application-defined scheduling and synchronization mechanisms the users will be able to cope with the complexity in designing new scheduling algorithms.

3 The shadow mechanism

The typical way of thinking of a mutual exclusion protocol is to consider it together with the implementation of the scheduling algorithm. Typically, when implementing a mutual exclusion protocol (e.g., Priority Inheritance), the mutual exclusion callbacks know the internal data structures of a particular implementation of a scheduling algorithm (e.g,, the Rate Monotonic ready queue) so they can modify it to implement a desired behavior (e.g., removing a blocked

[1] In our proposal the task pointed by a shadow pointer consumes its own capacity.

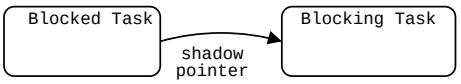

Figure 1. The shadow relationship can be thought as a pointer to a shadow task.

task from the queue and moving the task that inherits the priority out of the priority order).

This approach is commonly used because some protocols directly interact with the scheduler, making the protocol dependent on the particular scheduling algorithm implementation. Although a solution based on a direct interaction between the scheduler and the resource protocol is efficient in terms of runtime overhead, it limits the full modularity of the particular scheduler implementation, preventing the substitution of a scheduling algorithm with another that handles the same task and synchronization models. For example, the Rate Monotonic scheduler could be replaced by Deadline Monotonic, since both are still compatible with Priority Inheritance. However, in general the substitution cannot be done if the implementation of Priority Inheritance depends on the particular data structures used in the scheduling algorithm.

The fundamental idea of our approach is to replace the queue reordering, needed for the implementation of most resource access protocols, with a more general interaction method[2]. Such a mechanism is based on the concept of *shadow tasks*.

A shadow task is a task that is scheduled in place on another task chosen by the scheduler. When a task is blocked by the protocol, *it is kept in the ready queue*, and a shadow task is binded to it; when the blocked task becomes the first task in the ready queue, its binded shadow task is scheduled instead. In this way, the shadow task "inherits" the priority of the blocked task. The shadow relation can be thought as a pointer from the blocked task to the blocking task (see Figure 1).

When a task τ releases a mutex, all the shadow pointers that were set to τ (because the corresponding tasks were blocked on the resources) have to be voided, and the system has to check again for preemption. In fact, the tasks that had the shadow pointer reset may have higher priority than the task that released the resource, hence τ needs to be preempted. In general, the task that will be scheduled after unlocking depends on the particular application-defined scheduling policy.

[2]Our approach does not handle direct blocking on a synchronization point. That feature is usually part of every application defined scheduling interface and it is needed to implement blocking primitives that have no relations with mutual exclusion.

In the general case of nested resources, a directed acyclic graph (DAG) can grow, which expresses the dependencies between the tasks in the system (see Figure 3). It is worth noting that, in the classical approach, implementing the transitive property of priority inheritance requires at least some bookkeeping. On the other hand, under the shadow mechanism, handling transitive blocking is much simpler, since transitivity is expressed by the graph. Also note that, since the dependency relations are encoded into the DAG graph, deadlock conditions can be easily detected simply checking for cycles every time a shadow pointer is set.

Common mutual exclusion protocols like Priority Inheritance, Priority Ceiling and Stack Resource Policy can be implemented using the shadow mechanism, as shown in the next section. Since the shadow mechanism is independent of the implementation of a particular scheduling algorithm, mutual exclusion can be performed globally among tasks handled by different scheduling algorithms (for example, we can enable resource sharing using Priority Inheritance between a task scheduled by Round Robin and a task scheduled by EDF).

4 A prototype implementation

An important issue when implementing a scheduling mechanism is to keep the runtime overhead as low as possible. In this section we will describe how the shadow mechanism has been successfully implemented in the S.Ha.R.K. kernel. Then, in the next section we will shortly describe the requirements that a kernel should fulfill to properly implementing the mechanism.

The S.Ha.R.K. kernel internally maintains a task control block, called `proc_table[]`, that stores all the information about a task. The task identifier coincides with its index inside the table, `NIL` being an invalid task ID constant. The running task is stored in a global variable called `exec_shadow`, whereas the task selected by the scheduler is stored in the `exec` variable (note that it may be different from `exec_shadow` due to the shadow mechanism).

A new field `shadow` is added to the task control block. This field points to the shadow task. Initially, the shadow field is equal to the task ID (no substitution; see Figure 2.a). When the task blocks, the shadow field is set to the task ID of the blocking task, or to the task that must inherit the blocked task priority (see Figure 2.b). As noted in Section 3, a graph can grow during multiple resource conflicts (see Figure 3).

Then, the scheduler is decoupled from the dispatcher. In S.Ha.R.K., that is obtained using two different function calls, one for the scheduler and one for the dispatcher, that can be customized to imple-

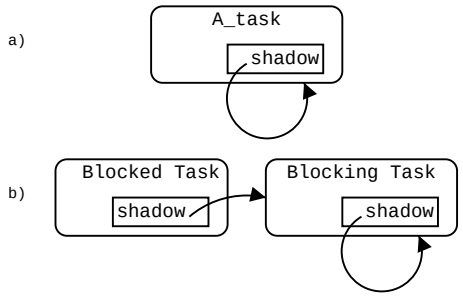

Figure 2. The implementation of the shadow mechanism. a) typically, the shadow field points to the task itself. b) the shadow pointer is set when a task blocks.

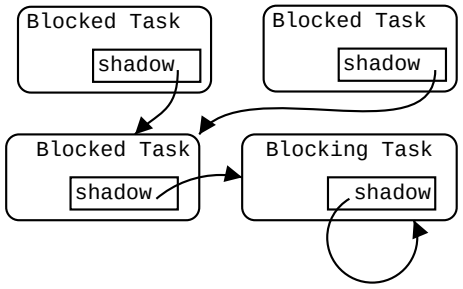

Figure 3. Using the shadow mechanism, a graph can grow independently from the data structures used by the scheduling module.

```
extern taskID_t exec, exec_shadow;
[...]
exec_shadow = exec = appl_defined_scheduler();
while (exec_shadow != proc_table[exec_shadow].shadow)
  exec_shadow = proc_table[exec_shadow].shadow;
appl_defined_dispatch(exec_shadow);
[...]
```

Figure 4. The scheduler code.

```
while (mutex->owner != NIL) {
  proc_table[exec_shadow].shadow = mutex->owner;
  <record the task as ''blocked''
   inside the mutex structure>
  scheduler();
  <context change>
}
mutex->owner = exec_shadow;
```

Figure 5. The Priority Inheritance lock callback.

ment application defined scheduling inside the kernel. The source code is tiny, and in our implementation it looks like the one showed in Figure 4.

In the following sections we show how the most common synchronization protocols are implemented.

4.1 Priority Inheritance

The implementation of the Priority Inheritance Protocol using the shadow mechanism is straightforward, because the shadow mechanism itself allows a natural expression of an inheritance protocol. Considering the lock an unlock callbacks, we have the following implementation.

lock When a task blocks, its shadow pointer is set to point to the blocking task. The callback also stores the information about the fact that the task has blocked on the mutex (that information will be used later in the unlock callback). A pseudo code of the callback is shown in Figure 5.

Please note that the lock must be done inside a while loop. That is because when all the shadow pointers of the blocked tasks are reset to unlock them, all them become suddenly ready, and only the first ready task will access the resource. The other tasks have to wait their turn blocking again. However this condition is rare, because tasks are scheduled using some kind of order (e.g., a priority), so every unblocked task will access the resource in the given right order.

unlock When a task unlocks a mutex, if some tasks are blocked on it the shadow pointer are reset. Then, the system is rescheduled to allow the unblocked tasks to preempt the running task.

task			a	b	c	d	e	f	g	h
preemption level			1	2	2	2	3	4	4	5
sysceiling		curr	shadow pointer							
1)	0	nil	a	b	c	d	e	f	g	h
2)	2	d	a	a	a	a	e	f	g	h
3)	4	g	f	a	a	a	f	f	f	h

Table 1. The SRP Implementation using the shadow mechanism.

4.2 Priority Ceiling

The implementation of the Priority Ceiling (PC) protocol can be done using the original proposal from Rajkumar in [6]. In this case, S.Ha.R.K. maintains a queue of the blocked mutexes ordered by ceiling. The while condition in Figure 5 becomes the PC blocking condition, and the shadow pointer in case of blocking is set equal to the owner of the locked mutex with the highest ceiling.

When that mutex is unlocked, all the shadow pointers of the blocked tasks are reset, letting the blocked tasks to test again the blocking condition.

4.3 SRP/Immediate Priority Ceiling

The implementation of the Stack Resource Policy (SRP) using the shadow mechanism is a little bit more complex than the other presented above, because the SRP protocol does not "directly" use the inheritance. Please also note that the implementation of the Immediate Priority Ceiling protocol *is* the same as the SRP implementation, because the only difference between the two is the queuing policy, and the shadow mechanism does not rely on that.

Basically, all the tasks that use SRP have a preemption level (that can be detached from the actual application-defined scheduling), and they are inserted in an ordered list, called *tasklist*. When a task locks a mutex and changes the system ceiling, all the shadows pointers of the tasks with preemption level less than the current task are set to the locking task, and viceversa when a mutex is unlocked.

The real algorithm is slightly optimized. For example, consider a task set of 8 tasks. For each task $\tau_a \ldots \tau_h$, we represent the shadow pointer and its preemption level. There are also two global fields: *current*, used to scan the tasklist (it always points to the last task that has a ceiling equal to the system ceiling), and *sysceiling*, that stores the system ceiling.

When the system starts, the situation is as depicted in Table 1 row 1).

Suppose that τ_a is scheduled and locks a mutex that causes sysceiling to become 2. The situation will be the one depicted in Table 1 row 2). Now suppose that task τ_f preempts task τ_a (the shadow setting does not change). Then, suppose that task τ_f locks a mutex that raises sysceiling to 4. The shadows will be set as in Table 1 row 3).

In practice, the system maintains a stack of the locked mutexes. Each mutex has in its descriptor the space for implementing a stack, useful in the unlock() function to undo the modifications to the shadow pointers done with the last lock(). This approach minimizes the number of shadows to be set, so minimizes the complexity of the lock/unlock operations.

Note that this implementation creates a tree in the shadows pointers (i.e., when sysceiling=4, task τ_c points to task τ_a that points to task τ_f). This may cause a performance a little worse with respect to a "one-jump" shadow set. Anyway, this is not a big problem because when a task is preempted it is very unlikely that it may be rescheduled before the end of another high priority task, so the multiple jumps in the shadow pointers appear rarely.

5 Toward a generic interface

In order to be used in real systems, the shadow mechanisms should be supported by a generic implementation-independent interface that allows the user to specify the desired behavior without depending too much on platform-dependent aspects.

The following paragraphs show the modifications that must be made to implement the mechanism in a generic system.

Scheduling separated from Dispatching. The first thing behind the shadow mechanism is the separation between the behavior of the scheduler (that is "choose the task that must be executed now") from the behavior of the dispatcher (that is "hey, someone has decided that this task will be executed next"). If you think for example at a scheduler implementation with a ready queue, the scheduling would be "look at the first task in queue", whereas the dispatching would be "remove the first task from the queue".

Sharing resources under temporal isolation. When the kernel supports temporal isolation [1], the mechanism for handling the execution budget allocated to each task may influence the blocking time experienced by tasks. In fact, if a task that causes blocking finishes its budget while executing in a critical section, the blocking time is affected by the budget replenishment policy.

As a consequence, when sharing resources between scheduling algorithms that support temporal isolation, it is important to consider how budget is handled. Using the shadow mechanism, it can happen that a task is dispatched not because it is scheduled by its application defined scheduler, but because some other task's shadow pointer is pointing to it. In this case, the budget exhaustion check should be disabled,

and a proper accounting of the consumed capacity should be done, to let the pointed task terminate its critical section. This fact can lead to tolerate negative budgets, because in general it is not known when a task is going to access a critical section. A theoretical analysis of blocking time and negative capacities can be found in [2]. Please also note that negative budgets are often inevitable due to non deterministic behaviors of the hardware, so it is something that the scheduling algorithms often have to care about anyway.

Handling the shadow chain. When using the shadow mechanism, between the scheduling and the dispatching operations, the kernel has to follow the shadow chain to get the task that is dispatched next. From our experience with the Shark kernel we can say that the shadow DAG rarely is deeper than a few levels, so following the shadow chain usually takes a negligible execution time.

An interface for mutual exclusion. Existing application-defined schedulers are only marginally influenced from the adoption of the shadow mechanism, because they simply schedule tasks without looking at what happens to the shadow pointers.

When implementing the shadow mechanism, the set of callbacks related to the scheduling and the set of callbacks related to the mutual exclusion should be clearly separated.

In particular, the separation between the callbacks relative to the scheduling from those relative to mutual exclusion is useful when integrating a mutual exclusion mechanism at the global level and not only from a single application defined scheduling. Moreover, the separation between the two interfaces helps the developer to focus on what he has to do, without being distracted by other parts that can be handled in an independent way.

Finally, a primitive should be added to modify the shadow pointer of a given task. The function can be used to set/reset a shadow pointer, and possibly to do a deadlock prevention check (the function should simply verify that no cycles are created in the shadow DAG graph).

6 Conclusions

This paper presented a novel method, the shadow mechanism, for implementing resource access protocols independently of the data structures defined in application-defined schedulers. The proposed mechanism can be used to make the mutual exclusion protocols orthogonal to the scheduler implementation, helping the designer to reuse the developed code with different scheduling algorithms.

The shadow mechanism has been successfully adopted in the S.Ha.R.K. kernel with negligible overhead for implementing classical resources access protocols, such as Priority Inheritance, Priority Ceiling, and Stack Resource Policy. Samples of the real S.Ha.R.K. code have been also illustrated, and an informal proposal for a generic interface for supporting the mechanism has been discussed. We believe that the shadow mechanism can be supported by other application-defined scheduling interface with little effort and great benefits.

References

[1] L. Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[2] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, London (UK), December 2001.

[3] B. Ford and S. Susarla. Cpu inheritance scheduling. In *Proceedings of OSDI*, October 1996.

[4] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[5] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *IEEE Proceedings of the Real-Time Systems Symposium*, London (UK), december 2001.

[6] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, chapter 3. Kluwer Academic Publishers, 1991.

[7] M. A. Rivas and M. G. Harbour. Posix-compatible application-defined scheduling in marte os. In *Proceedings of 14th Euromicro Conference on Real-Time Systems*, pages pp. 67–75, Vienna, Austria, June 2002.

[8] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.

[9] P. Valente and L. Rizzo. A wf2q+ scheduler implementation in freebsd. http://info.iet.unipi.it/~luigi/ps_sched.