Controling Kernel Scheduling from User Space: an Approach to Enhancing Applications' Reactivity to I/O Events

Vincent Danjean and Raymond Namyst

LaBRI, Université de Bordeaux I, F-33405 Talence, France.

Abstract. In this paper, we present a sophisticated mechanism that allows an application to tightly control the way I/O events are handled within the underlying operating system's kernel. The goal is to provide an efficient user-level interface to allow applications to quickly detect and handle asynchronous I/O events. Reactivity is obviously a crucial property for many time-critical applications. For instance, it is a major issue in distributed applications where poor reactivity to communication events may have a direct influence on the observed latency of communications, and may even impact on the whole application behavior. Our approach is based on an extension of the *Scheduler Activation* kernel scheduling model, originally proposed by Anderson et al. The idea is to minimize the time elapsed between the detection of an I/O event within the kernel and the scheduling of the appropriate user-level thread within the application. The progress of I/O operations still takes place in kernel space, but the scheduling decisions are taken in user space. We have implemented the core of our proposal within the Linux kernel, and the PM^2 multithreading environment was ported on top of the corresponding user-level API. Our experiments demonstrate that the implementation of our kernel extensions is very efficient and that the overhead introduced by the user-level interface is very small. Moreover, the observed results show that our approach competes favorably against approaches based on the use of signals or kernel threads.

1 Introduction

In the past few years, the widespread use of clusters of SMP workstations for parallel computing has lead many research teams to work on the design of portable multithreaded programming environments [1, 2]. One major challenge is to design environments that reconcile the portability and efficiency requirements of parallel applications: applications have to be portable across a wide variety of underlying hardware while still being able to exploit much of its performance. Most noticeably, many efforts have been focused on the problem of performing efficient communications in a portable way [3, 4] on top of high-speed networks [5, 6].

However, one major property has often been neglected in the design of these distributed runtime systems: the *reactivity* to communication events. We call *reactivity* of an application its ability to handle external, asynchronous events as soon as possible within the course of its regular execution. Trying to minimize (or at least to bound) the response time to a network event is indeed a critical parameter for many applications since the observed *latency* of messages may become proportionally long. Actually, many distributed environments suffer from this problem since the performance of mechanisms involving synchronous interactions between nodes directly depends on their reactivity to incoming communications. A Distributed Shared Memory (DSM) subsystem, for instance, may behave very poorly if the internal request messages exchanged to fetch remote data are not handled as soon as they reach the destination node[7]. MPI can also be subject to such bottlenecks when using asynchronous primitive such as asynchronous sends (`isend()`)[8] or remote get and put operations (MPI2). Most communication schemes including a back and forth interaction with a remote agent, such as a remote procedure call, are extremely sensitive to the reactivity of the peer.

In this paper, we investigate the problem of maximizing the reactivity to I/O events in a user-level multithreaded context, focusing on I/O events detected in kernel space, such as those notified by hardware interrupts (see [9] for a discussion about detecting I/O events in a reactive manner by using a user-level polling-oriented policy). More precisely, our goal is to propose a mechanism able to quickly propagate kernel I/O events to the application. That is, we try to minimize the time elapsed between the detection of an I/O event (typically notified by a hardware interrupt) and the execution of the appropriate user-level handler within the application. This mechanism is an extension of the *Scheduler Activations* model introduced by Anderson et al [10]. A flexible user-level API allows the application to tightly control the scheduling of threads when I/O events are detected, thanks to a sophisticated cooperation between the kernel and user space applications.

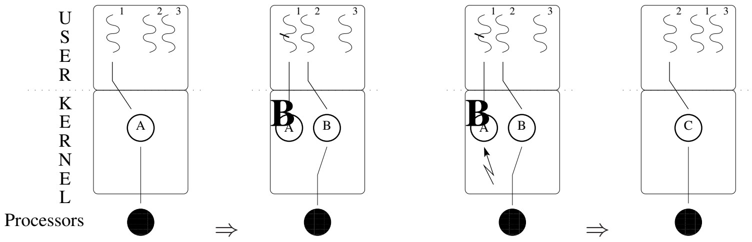
2 Dealing with I/O events in a multithreaded context

To understand how an I/O event is typically handled within a multithreaded application, let us start from the point where I/O events are actually detected: inside the operating system's kernel. For instance, let us consider an I/O event notified by an hardware interrupt. For obvious security reasons, hardware interrupts cannot be handled directly in user-space (at least in traditional operating systems). Indeed, the occurrence of such an interrupt usually triggers the execution of an interrupt handler within the operating system's kernel. Then, the corresponding event may in turn be notified to a process (or kernel thread) either asynchronously or synchronously.

In the second case, a process (or kernel thread) can be notified synchronously by performing a blocking system call, sleeping inside the kernel until the appropriate event occurs. The triggering of the corresponding interrupt will simply wake up the process and, later, let it return back in user space. This mechanism has the advantage to be extremely simple from the application's point of view. However, its efficiency depends on the capability of the thread scheduler to execute the kernel thread as soon as it is woken up, especially if the number of threads in "ready" state exceeds the number of processors.

The *Scheduler Activations* scheduling model, proposed by Anderson et al [10], features a set of mechanisms that enable the kernel to notify a user-level process whenever it makes a scheduling decision affecting one of the process's threads. It is implemented as a set of *up-calls* and *down-calls*.

A traditional system call is a *down-call*, from the user-level down into a kernel-level function. The new idea is to introduce a corresponding *up-call*, from the kernel up into a user-level function. An up-call can pass parameters, just as system calls do. An *activation* is an execution context (*i.e.*, a task control block in the kernel, similar to a kernel-level thread belonging to the process) that the kernel utilizes to make the up-call. The key-point is that each time the kernel takes a scheduling action affecting any of an application's threads, the application receives a report of this fact and can take actions to (re)schedule the user-level threads under its control. Of course, if several events occur at the same time, only one upcall is done to notify all the events. Figure 1 illustrates how a blocking system call is handled by such a system.



(a) Running ⇒ (b) Blocking syscall (c) Waking up ⇒ (d) Running

1(a) The user-level thread 1 is running onto an activation A bounded to a processor.
1(b) The thread 1 makes a system call blocking the activation A. Another activation B is launched which runs another user-level thread 2.
1(c) An interrupt occurs waking up the blocked activation A.
1(d) A new activation C is created to send the unblocking notice event. To run this activation, the activation B is preempted. The activation C can chose which user thread to run. Activations A and B can be discarded once the state of their thread has been sent to the application in the upcall.

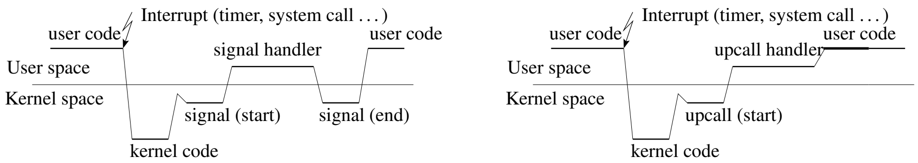
Fig. 1. Blocking System Call

There are several issues with the original Scheduler Activation model that make it difficult to obtain an efficient implementation[11]. First, a lot of information needs to be sent by the kernel to the application. Each time a preemption or an unblocking notice occurs, the kernel needs to send to the application the state of the preempted or unblocked thread. When a thread executes a blocking system call and resumes, the application needs to manage (copy) those information several times. Another difficulty comes from the fact that the application has no control at all over the number of virtual processors assigned to it (it only knows the number of currently assigned virtual processors). This means that the application (in particular upcalls) has to be made completely reentrant. Finally, the fact that the upcalls tell the application about the state of the interrupted (or unblocked) thread makes this model unable to handle internal kernel scheduling state. A blocked thread has to run in kernel mode until it becomes ready to come back in user mode before being signaled as unblocked for the application.

3 Our Proposition: an Efficient and Reactive Scheduling Model

3.1 A New SA-like Scheduling Model

To solve the aforementioned issues, we propose an extension of the Scheduler Activation model. The key idea is that the operating system kernel can not change the number of processors assigned to an application during its execution anymore. If a thread blocks, the kernel still has to create a new activation to give back the virtual processor to the application. However, the kernel is not allowed to preempt a virtual processor on its own anymore. More precisely, if the kernel preempts an activation of an application (as this often occurs in a time-shared operating system such as Linux), the application is simply left unnotified.



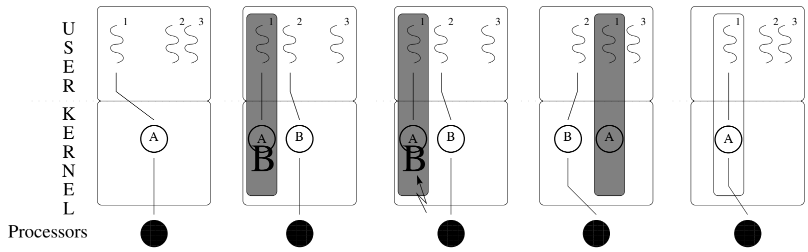
(a) Signal Flow (b) Upcall Flow

2(a) At the end of the signal handler, a system call switch to kernel mode to restore the user environment.
2(b) At the end of the upcall, the user library is able to directly continue the interrupted user code. The upcall handler must take care to restore the proper environment, but no system call is needed anymore.

Fig. 2. Restarting After an Event

In our model, upcalls are always performed within the currently running activation, like Unix signals. However, at the end of the upcall, the application does not go back into the kernel using a system call as it is done for signals. Thus, our upcalls are very cheap: there is only the addition of the execution of a user-space function with respect to a standard kernel/user space transition. Figure 2 shows our mechanism used to launch an upcall and Figure 3 illustrates how a blocking system call is handled by our new system. The kernel/user interface features several upcalls used by the kernel to notify the application. Obviously, a few system calls are also needed by the application to interact with the kernel.

Upcalls The main originality of our model is that the user-level thread scheduler has a complete control over the scheduling of threads, even at times they block or wake up within the kernel. This explains why the upcalls performed by the kernel in our model are slightly different from the ones in the original model. Two kinds of events may be reported by the kernel. *Synchronous events* are reported by upcalls raised synchronously with respect to the currently running thread. That is, the upcall is raised when the thread can not follow its standard flow of execution. Such upcalls typically tell the application that the current thread has just blocked (upcall **block**) and indicate if a new activation has been created or if a previously unblocked one has been restarted. On the opposite,



(a) Running (b) Blocking syscall (c) Waking up (d) Running (e) Restarting

3(a) The user-level thread 1 is running onto an activation A bounded to a processor.
3(b) The thread 1 makes a system call blocking the activation A in the kernel. Another activation B is launched which runs another user-level thread 2.
3(c) An interrupt occurs. The blocked activation A is now ready to wake up (but it will only wake up when requested by the application).
3(d) The activation B is used to send the unblocking notice event with an upcall. The user thread scheduler puts the thread 1 in the ready-to-run thread queue. Its register state is unknown to the application: the thread is always in the kernel.
3(e) When the user thread scheduler wants to restart the unblocked thread, it saves the currently running thread state and discards the current activation in favor of the unblocked thread activation.

Fig. 3. Blocking System Call

asynchronous events are reported by upcalls that behave like Unix signals. For instance, the **unblock** upcall is used to notify the application that some previously blocked activations can now be resumed. Thus, the application can choose which one is to be restarted immediately.

System Calls An application can use several system calls to use and control the Scheduler Activations mechanism. The **init** system call is performed at initialization time to notify the kernel that the application will use the Scheduler Activations mechanism. The kernel is also told about the addresses of the user-space upcalls entries and the addresses of some variables accessed by the kernel. The **revoke** system call tells the kernel that the application does not need the underlying virtual processor anymore until either the application requests this virtual processor again or a blocked activation wakes up. The **request** system call is used by the application to tell the kernel that it wants to use (again) a virtual processor that has been revoked. Finally, the **wakeup(`act_id`)** system call destroys the current activation so that the unblocked activation of identifier `act_id` can be resumed within this virtual processor.

3.2 A User-Level Interface to our SA Model Extension

Obviously, such a complex scheduling mechanism is not intended to be used directly by regular applications. Therefore, we developed a stack of software layers to meet the needs of a wide range of applications. The overall design of our implementation is shown on figure 4.

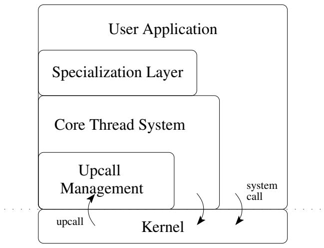


Fig. 4. User Interface Design

The **Core Thread System** is the most important part of our thread package. It is responsible for providing the core locking mechanisms, for managing thread related data structures (thread control blocks) and for managing thread states. It also provides a basic scheduling policy (round robin) that is used by default for scheduling decisions. The **Upcall Management layer** handles all events sent by the kernel to our thread package by the means of upcalls. These functions modify the thread states when needed. For instance, this layer is responsible for removing blocked threads from the ready queue, and for inserting unblocked threads back into this queue. The **Specialization Layer** is intended to be used by the application to customize the behavior of the core thread system. In particular, several scheduling functions can be redefined very easily, such as the function responsible for choosing the next thread to run each time a context switch takes place (`find_next_to_run`). So, virtually any kind of scheduling policy can be implemented on top of our software stack, because the control over the scheduling of kernel activations is entirely done in user space.

The Core Thread System Interface The core thread system is mainly used by the specialization layer, though it may also be used directly by the application. It features all basic thread operations such as thread creation, destruction, synchronization, *etc.* These functions are often extended in the specialization layer in order to manage more attributes or to offer more advanced features (synchronization barriers, *etc.*). It also provides some low-level entry points to manage the core thread system. One of the most important is the `thread_find_next_to_run` function invoked by the core thread system each time a context switch is performed between user-level threads.

This has been inspired by the OpenThread interface[12] where the application has the ability to manage thread queues and to register call-back functions for each thread state transition. However, the fact that our system can manage more than one processor (ie real parallelism) makes this interface not usable as-is. For example, the locking scheme can not be the same if the application wants one run queue per virtual processor or one for the whole application. So, for now, such customizations, though possible (and actually implemented), require deep modifications within the library.

4 Implementation Issues

We now discuss the main issues we had to address during the implementation of our model, focusing on the issues related to our extension of the Scheduler Activation (SA) model. Issues related to the original model are discussed in [10].

4.1 On the number of Kernel Activations

In the original SA model, an unblocked activation is destroyed as soon as the corresponding event has been notified to the application. In our model, as we do not transmit the state of blocked threads to user mode anymore, the activation cannot be destroyed at this point. We must wait for the application to request a restart of the unblocked activation. However, this new model does not require more activations than the original. The activations can live longer, but the kernel creates the same number of activations as in the original model. In particular, when an activation blocks, the kernel tries to restart an unblocked one (if available) instead of creating a new one. So, new activations are created only when a virtual processor becomes free while all activations are either in "running" or "blocked" states.

For now, there is no limit in our implementation about the number of activations that can be created by the kernel. If each user thread executes a blocking system call, then there will be as many [blocked] activations created by the kernel as user threads. To limit kernel resources, we could make blocking system calls either fail or not generate an upcall. However, this leads to other problems: what should be done if all blocked threads need the action of another user thread (not launched due to resource limit on the number of activation) to unblock. Lack of activation could also lead to a special signal so that the application can try to unblock some of its threads or exit with an error code. However, all these behaviors are different from the one of a standard thread library. So, no limit is implemented and, in the bad case, our model can consume as many kernel resources as in the current standard pure-kernel thread library.

4.2 Critical Sections and Upcall Reentrancy

The problem of concurrency between upcalls and user-level code is a hard one, and there are a number of situations where upcalls should be delayed because their execution may conflict with some currently running user-level execution flow, or even with another upcall within the same virtual processor. We solved these problems by implementing a cheap per-virtual processor mutual exclusion mechanism accessed by both user-space and kernel-space. Each virtual processor is equipped with its own lock variable, which is allocated in user-space. Of course, the kernel is told about the location of each of these locks at initialization time.

Just before executing an upcall on a virtual processor, the kernel sets the variable. The user-level core has the responsibility to clean this variable at the end of each upcall. So, when the kernel wants to make an informative upcall while this variable is set, the upcall is delayed. In the case the kernel wants to make an upcall because the previous activation has just blocked, then the event is simply not reported to the application and thus the previous blocked activation is still considered as running. When the activation unblocks, it resumes as if it never stopped. This case (blocking during an upcall) should be rather rare but needs to be correctly handled. We cannot launch yet another upcall if the previous one has been interrupted because the upcall code has been swapped out.

5 Performance

All the features described in this paper were implemented within the Linux kernel (v2.4.19). We now present an evaluation of the cost of some raw kernel mechanisms we did implement. This can be viewed as the overhead of our mechanism compared to a pure kernel-level scheduler.

5.1 The Cost of Kernel Mechanisms

Performing an Upcall. To perform an upcall, the kernel only needs to write a few parameters on the user stack (previous stack pointer and program counter, virtual processor number, ...) before returning as it usually does. On the application side, the upcall handler is executed and then uses a mini assembly trampoline to return to the previously interrupted code. In table 1, one can see the time lost by the application when the kernel preempts the application and executes an upcall, a signal handler or nothing (just returns after regular kernel mode processing). These measures have been taken on an Intel P4 1.8GHz processor. All numbers were gathered from multiple runs. The initial run is not timed to reduce cold cache miss effects. One can observe that upcalls and signal handlers add a few overhead with respect to a regular return. However this overhead is smaller for an upcall than for a signal handler. This is due to the fact that the kernel does not need to save and restore all registers for an upcall.

	Time lost	Overhead
Regular path	13.1 μs	0 (+0 %)
Upcall	14.1 μs	+1.0 μs (+7.63 %)
Signal handler	16.5 μs	+3.4 μs (+25.95 %)

Table 1. Upcalls Overhead

`upcall overhead`	1.0 μs
`getpid()`	0.92 μs
`act_cntl(RESTART_UNBLOCKED, id)`	2.17 μs

Table 2. Restart Overhead and System Calls

Restarting an Unblocked Activation. As explained earlier, the restart of an unblocked thread requires a system call in our model, because the underlying activation may be suspended deep in the kernel. Such a system call (`act_cntl(ACT_CNTL_RESTART_UNBLOCKED, id)`) discards the current activation and restarts the unblocked one (with the id `id`) on the current virtual processor. In table 2, one can see the time needed by a basic system call (`getpid`) and the time needed to restart an unblocked activation. For the latter, we measure the time from the beginning of the system call to the effective restart of the unblocked thread: this time includes the system call and the upcall generated when the unblocked activation restarts. So, the overhead due to discarding the previous activation and switching to the awakened one costs only $0.25\,\mu$s.

5.2 Efficiency and Flexibility

We now evaluate the overhead of our Scheduler Activation model within a user-level thread scheduler, using both a POSIX-compliant API and the API of the PM^2 distributed programming environment [2], which was ported on top of our software stack. Table 3 shows the time needed to perform a context switch between user-threads, and the execution time of a synthetic program (`sumtime`) that creates, synchronizes and destroys a large number of threads (it makes a sum of the n first integers recursively, each thread managing half of the interval of its father).

	`yield()`	`sumtime(2000)`
`libpthread`	$1.16\,\mu$s	7215 ms
`POSIX`	$1.01\,\mu$s	12 ms
`PM`^{2}	$0.40\,\mu$s	12 ms

Table 3. Basic Thread Operations

We can see that in any case, our user-level libraries perform a lot better than the Linux kernel thread library, while keeping the ability to use SMP machines and correctly manage blocking system calls). The POSIX thread version does not improve a lot the `yield` operation since it is not a pthread call but a system call (`libpthread` does not have `pthread_yield` operation, `sched_yield` is used instead). The little improvement comes from the fact that there is only one running kernel thread with our POSIX library (so no real kernel context switch is performed) while there are two with the `libpthread` library (one for each application thread). If we redefine the `sched_yield()` call as an alias for our `thread_yield()` function in the `POSIX` library, then POSIX and PM^2 libraries show similar performance for `yield()` operations. With the `sumtime` program, the lost of performance with the standard `libpthread` library worsens when the number of threads grows. In this example, the program manages about 2000 threads. With our thread libraries, we can easily use this program with more than 100 000 threads. It is not possible with the standard `libpthread` library.

6 Conclusion and future work

In this paper, we present the design and implementation of a fast and flexible extension to the Scheduler Activation model, together with a user-level software stack to allow an easy customization of the thread scheduling policy. The application has a complete control over this scheduling because all decisions are made in user-space. The kernel mechanisms were fully implemented within Linux (v2.4.19). A POSIX-thread compliant user-level thread library has been developed on top of our software stack, as well as the PM^2 distributed multithreaded programming environment. We conducted a number of experiments which show that the core mechanisms are very efficient, and that the overhead introduced within the user-level schedulers is very small.

We plan to measure the impact and the improvement due to the use of these mechanisms within large multithreaded networking applications. We also intend to investigate the tuning of specific scheduling policies in the PM^2 environment, as far as network I/O operations are concerned. The goal is to maximize the reactivity of the threads in charge of handling network events so as to speed up the whole distributed application.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The Nexus approach to integrating multithreading and communication. Journal of Parallel and Distributed Computing **37** (1996) 70–82
2. Namyst, R., Méhaut, J.F.: PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In: Parallel Computing (ParCo '95), Elsevier Science Publishers (1995) 279–285
3. Prylli, L., Tourancheau, B.: BIP: A new protocol designed for High-Performance networking on Myrinet. In: 1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98). Volume 1388 of Lecture Notes in Computer Science., Orlando, USA, Springer-Verlag (1998) 472–485
4. Aumage, O., Bougé, L., Méhaut, J.F., Namyst, R.: Madeleine II: A portable and efficient communication library for high-performance cluster computing. Parallel Computing **28** (2002) 607–626
5. Dolphin Interconnect: SISCI Documentation and Library. (1998) Available from URL `http://www.dolphinics.no/`.
6. Myricom: Myrinet Open Specifications and Documentation. (1998) Available from URL `http://www.myri.com/`.
7. Antoniu, G., Bougé, L.: DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In: Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01). Volume 2026 of Lect. Notes in Comp. Science., San Francisco, Held in conjunction with IPDPS 2001. IEEE TCPP, Springer-Verlag (2001) 55–70
8. Prylli, L., Tourancheau, B., Westrelin, R.: The design for a high performance MPI implementation on the Myrinet network. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Proc. 6th European PVM/MPI Users' Group (EuroPVM/MPI '99). Volume 1697 of Lect. Notes in Comp. Science., Barcelona, Spain, Springer Verlag (1999) 223–230
9. Bougé, L., Danjean, V., Namyst, R.: Improving Reactivity to I/O Events in Multithreaded Environments Using a Uniform, Scheduler-Centric API. In: Euro-Par, Paderborn (Deutchland) (2002)
10. Anderson, T., Bershad, B., Lazowska, E., Levy, H.: Scheduler activations: Efficient kernel support for the user-level managment of parallelism. In: Proc. 13th ACM Symposium on Operating Systems Principles (SOSP 91). (1991) 95–105
11. Williams, N.J.: An Implementation of Scheduler Activations on the NetBSD Operating System. In: USENIX Annual Technical Conference. (2002) 99–108
12. Haines, M.: On Designing Lightweight Threads for Substrate Software. In: USENIX Technical Conference, Anaheim, CA, USENIX (1997) 243–255