Integrating Kernel Activations in a Multithreaded Runtime System on top of Linux

Vincent Danjean[1], Raymond Namyst[1], and Robert D. Russell[2]

[1] Laboratoire de l'Informatique du Parallélisme
École normale supérieure de Lyon
46, Allée d'Italie
F-69364 Lyon Cedex 07, France
{Vincent.Danjean, Raymond.Namyst}@ens-lyon.fr

[2] Computer Science Department
Kingsbury Hall
University of New Hampshire
Durham, NH 03824, USA
rdr@unh.edu

Abstract. Clusters of SMP machines are frequently used to perform heavy parallel computations, and the concepts of multithreading have proved suitable for exploiting SMP architectures. Generally, the programmer uses a thread library to write this kind of program. Such a library schedules the threads or asks the OS to do it, but both of these approaches have problems. Anderson et al. have introduced another approach which relies on cooperation between the OS scheduler and the user application using *activations* and *upcalls*. We have modified the Linux kernel and adapted the Marcel thread library (from the programming environment PM^2) to use activations. Improved performance was observed and problems caused by blocking system calls were removed.

1 Kernel Support for User Level Thread Schedulers

The increasing popularity of clusters of SMP machines creates a need for multithreaded programming environments able to fully exploit such architectures. Indeed, the thread model naturally helps to make efficient use of all available processors and to overlap I/O operations with computations. Furthermore, threads are often considered as "virtual processors" and are targeted as such by compilers or runtime support systems for portability purposes. However, these runtime systems are built on top of thread libraries that do not all have the same properties, and thus do not provide the same functionalities. Moreover, these properties directly depend on how much control the thread scheduler has over the architecture's resources. There are two principle kinds of threads: user-level and kernel-level, each with its own advantages and inconveniences.

Efficiency is the main advantage of user-level thread libraries, whose scheduler is completely implemented in user space. Most operations on threads (creations, context switches, etc.) can be done without any call to the operating system. As a result, some computations utilizing these threads may perform one or two orders of magnitude better than kernel-level threads. Furthermore, user threads are much more efficient in terms of kernel resource consumption, which means there can often be many more of them per application. Finally, since user-level threads are implemented in user space, they can be tailored to each user's application. The disadvantage is that user-level threads are "ignored" by the OS and thus cannot be scheduled correctly in many cases. For instance, since user threads within the same process cannot be scheduled concurrently on multiple processors, no real parallelism can be achieved. Similarly, when a thread makes a blocking system call (for example, a `read()` on an empty socket), all the threads in that process are blocked.

Obviously, kernel-level threads do not suffer from these drawbacks, since their scheduling is realized within the OS kernel, which handles them the same way it handles processes, except that multiple threads may share the same address space. It is therefore possible on an SMP machine for the kernel to simultaneously assign processors to multiple threads in the same application, thus achieving true parallelism. Furthermore, when one thread makes a blocking system call, the kernel can give control to another thread in the same application. However, even if operations such as thread context switching are more efficient than those related to processes, they still require system calls to be performed.

1.1 The Marcel Mixed Thread Scheduler

To try to obtain the best properties of the two kinds of threads, some libraries mix them together: there are a fixed number of kernel threads each running a number of user threads. This approach retains the efficient scheduling of user threads, but is able to take advantage of parallelism between threads on SMP machines. One such library is Marcel[5], which was developed for use by PM^2[4] (*Parallel Multithreaded Machine*), a distributed multithreaded programming environment. Marcel delivers good performance by eliminating some features from the POSIX pthreads specification that are not useful for scientific applications (*e.g.*, per-thread signal handling). In addition, it supports multiple optimizations as well as dynamic thread migration across a homogeneous cluster. Marcel has been ported to a number of different platforms. It utilizes a fixed number of kernel threads, each managing a pool of user-level threads.

1.2 Better Support: Kernel Activations

Although the two-level version of Marcel achieves better performance than the earlier user-level version, it still suffers from some of the problems discussed earlier. The first problem is that when a user thread makes a blocking system call, the underlying kernel thread is stopped too. It is possible with a few blocking user threads to block all the kernel threads, thereby blocking the whole application, even if some other user threads are ready to run. Another problem is that even if Marcel can control the scheduling of user-level threads in each pool, it cannot do anything between the different pools. So, if thread A in pool 1 holds a lock and is preempted by the system, then when thread B in another pool wants the lock, it has to wait for the OS to give control back to pool 1 so that thread A can release the lock.

These problems could be avoided if the OS scheduler reported its scheduling decisions to the application. One mechanism to achieve this cooperation is based on the concept of *activations*, which was first proposed in an article by Anderson et al.[1] Its authors implemented this mechanism with the FastThread library on the Topaz system. However, this system is no longer running, and the sources were never released. All the terms (*activation*, *upcall*, etc.) used in this paper come from this article.

This mechanism enables the kernel to notify a user-level process whenever it makes a scheduling decision affecting one of the process's threads. This mechanism is implemented as a set of *upcalls* and *downcalls*. A traditional system call is a *downcall*, from the user-level down into a kernel-level function. The new idea is a corresponding *upcall*, from the kernel up into a user-level function. An upcall 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 upcall. 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 action to (re)schedule the user-level threads under its control.

We have modified the Linux kernel by adding activations and changing the existing kernel scheduler to use upcalls to report some scheduling events to the Marcel scheduler running in user space. Upcalls are mainly used to report that a new activation has been created, that an activation has blocked in a system call, that a previously blocked activation has just been unblocked, or that an activation has been preempted. We have also modified Marcel to utilize this mechanism efficiently, as discussed in the next section.

2 Marcel on Top of Linux Activations

The user-level Marcel thread scheduler utilizes the new mechanism as follows: Marcel begins by making an `act_new()` system call to notify the kernel that it wants to utilize activations. The scheduler provides parameters that include a vector of entry points for a fixed set of user-level management functions to which the kernel will make upcalls. Whenever the kernel makes a scheduling decision affecting any of this process's activations, such as creating, blocking or unblocking it, the kernel informs the process by choosing one of its activations and using it to make the appropriate upcall, such as `upcall_new`, `upcall_block`, or `upcall_unblock`. In order to guarantee exclusive access to management information while executing one of these functions, the kernel maintains an internal mutual exclusion lock that allows only one upcall at a time to be outstanding per process. Therefore, the management function must make an `act_resume()` system call to release that lock after making its management decision but before

Table 1. Upcalls made by the LINUX kernel to the user-level thread scheduler

Upcall	**Description**
`upcall_new`	a new activation is starting
`upcall_block`	an activation blocked
`upcall_unblock`	an activation unblocked. The scheduler has its state, so it can restart the activation's thread when it wants.
`upcall_preempt`	an activation was preempted. The scheduler has its state, so it can restart the activation's thread when it wants.
`upcall_restart`	Used by the kernel to make an upcall (*e.g.*, in response to an `act_send()` system call) when it has no scheduling event to report.

executing application specific code. If the kernel scheduler decides that an activation holding this lock should be preempted, the kernel will preempt another activation instead (via `upcall_preempt`) and will simply reschedule the original activation without an upcall.

Our implementation of the activations within the LINUX kernel is close to the one proposed by Anderson et al. It is described more fully in [2]. The next section presents some general characteristics that are referred to in the following sections.

The programming interface provides a few new system calls, and the targeted thread library must be prepared to handle several kinds of upcalls. Table 1 describes the upcall interface used by the kernel to notify the user thread scheduler about certain scheduling events.

2.1 How it works

Figure 1 illustrates how MARCEL uses activations to keep both processors on a dual-processor SMP platform actively executing application threads, even when some threads are blocked in the kernel.

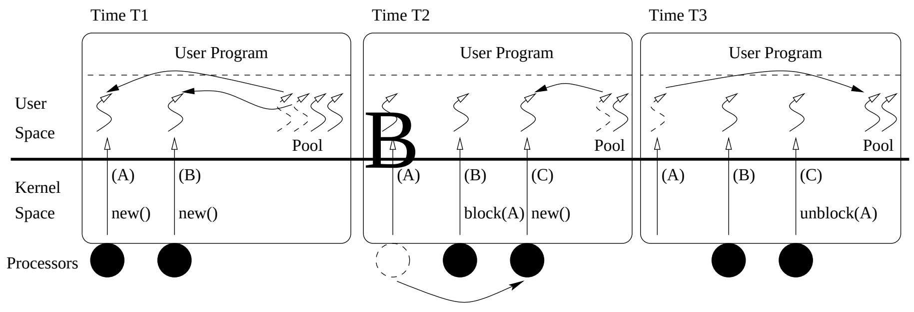

Fig. 1. A blocking system call with activations

At time T1, the kernel creates two activations "A" and "B" and makes an `upcall_new` to each. In each activation, the MARCEL scheduler will choose a ready application thread and give it control.

At time T2, the application thread running in activation "A" makes a blocking I/O system call. The kernel determines the process to which this activation belongs and creates a third activation, "C", into which it makes an `upcall_new`. In this activation, the MARCEL scheduler will choose a third application thread and then call `act_resume()` to release the mutual exclusion lock. The kernel next chooses one of the activations, say "B", and makes an `upcall_block` to it, providing "A" as the parameter to indicate which activation was just blocked. The MARCEL scheduler uses this information to keep track of the status of the corresponding application thread.

At time T3, the I/O request completes. The kernel then chooses one of the activations, say "C", and makes an `upcall_unblock` to it, providing "A" as the parameter to indicate which activation was just unblocked. The MARCEL scheduler now chooses whether to return the application thread previously assigned to "A" to the pool and continue running the application thread already assigned to "C", or vice versa. In either case, activation "A" remains idle until needed by the kernel to make another upcall.

2.2 Extensions to the original proposal

Although this work is mainly based on the *Scheduler Activation* model proposed by Anderson et al, we have developed a number of improvements which extend the set of supported system calls and increase efficiency in some situations.

One important point with activations is that the number of running activations at the application level is constant. In Anderson's implementation, this also meant that the number of activation structures for that user in the kernel was constant. This has the advantage of using a constant amount of kernel resources. However, it does not allow the kernel to handle blocking system calls properly, since a kernel activation structure is tied up during the time its thread is blocked, thereby preventing the kernel from running another user-level thread in that activation. Our implementation does not keep constant the number of activation structures for one user within the kernel. This allows us to handle any number of simultaneously blocking system calls, because whenever one activation issues a blocking system call, the kernel will create a new activation structure, if necessary, in order to keep constant the number of concurrently running activations at the application level. The cost of this is the additional kernel resources that are needed for the additional activation structures.

Several optimizations have been made to increase the performance of our implementation. When an activation blocks, we originally needed to make two upcalls: the first (`upcall_block`) to notify the application that an activation blocked, the second (`upcall_new`) to launch a new activation. This is now handled by only one upcall to `upcall_new`, which uses a parameter to tell the application whether another activation has blocked. An additional optimization has been made as far as preemption is concerned. In the original model, when an activation is preempted, an `upcall_preempt` upcall occurs, and an `upcall_new` upcall is made when the kernel is ready to restart an activation. Now, the application can tell the kernel at the end of the `upcall_preempt` upcall (with a parameter to the system call `act_resume()`) that instead of calling the upcall `upcall_new`, it can continue this activation directly.

2.3 Modifications to Marcel

Surprisingly, integration of Linux Activations within the Marcel library required almost no rewriting of existing code. We needed only a few localized extensions.

The major issue that we had to address was related to the "ready-threads" queue. The problem was to opt either for a global pool (as in a user-level version of Marcel) or for a collection of activation-specific local pools (as in the mixed version). We have opted for the global pool implementation because maintaining separate pools introduces a number of synchronization problems. In particular, when an activation gets blocked within the kernel, the other activations must retrieve the running threads that were kept in its ready-threads pool. Such a step requires a costly synchronization scheme and the associated overhead may become important in the presence of frequent I/O operations. The drawback of our strategy is that the global pool may become a bottleneck on a large number of processors.

Marcel uses a special lock to prevent concurrent access to its internal data structures. Our implementation of activations ensures that if the kernel preempts the Marcel thread which is holding this lock, then it is relaunched immediately (instead of the one running on the activation that receives the `upcall_preempt` upcall). This allows us to avoid contention situations in the presence of busy waiting threads. Note that a related problem can occur with the upcall `upcall_new`. Indeed, when a new activation is created, it may not succeed in acquiring the aforementioned lock. Since it is mandatory to run a regular Marcel thread when calling `act_resume()`, the activation must schedule a "dummy" thread. To this end, we have added a pool of preallocated "dummy" threads (together with their stacks) into Marcel.

3 Performance and Evaluation

The new version of Marcel on top of Linux Activations is completely operational, although we did not yet implement all the optimizations we discussed in the previous sections. To investigate the gain or the overhead generated by activations and upcalls, we have compared the new version of Marcel to the two existing versions (one purely user-level, one mixed two-level) as well as to native Linux kernel-level threads [3]. The tests were run on an Intel Pentium II 450 MHz platform running Linux v2.2.13. On this platform, we ran a microbenchmark program to measure the time taken by an *upcall* from the kernel up to user-space. This test reported an average time of $5\mu s$ per upcall.

Table 2. Performance of various thread libraries

Library	Single processor		Dual processor
	Basic	With I/O	With computation
MARCEL user-level	0.308ms	119.959ms	6932ms
MARCEL mixed two-level	0.435ms	23.241ms	3807ms
MARCEL with activations	0.417ms	10.118ms	3551ms
LINUXTHREAD (kernel-level)	13.319ms	14.916ms	3566ms

The test programs used to compare these libraries are all based on a common synthetic program. The basic program implements a *divide and conquer* algorithm to compute the sum of the first N integers. At each iteration step, two threads are spawned to compute the two resulting sub-intervals concurrently, unless the interval to compute contains one element. The "parent" of the two threads waits for their completion, gets their results, computes the sum and, in turn, returns it to its own parent. This program generates a tree of threads and involves almost no real computation but a lot of basic thread operations such as creation, destruction and synchronization.

In order to evaluate the different thread libraries in the presence of blocking calls, we have extended the previous program so as to make extensive use of UNIX I/O operations. In this case, we have simply replaced all the thread creation calls by a write into a UNIX *pipe*. At the other end of the pipe, a dedicated server thread simply transforms the corresponding requests into thread creations.

Finally, we also extended the basic version of the program by adding some artificial computation into each thread so that some speedup can be obtained on a multiprocessor platform.

3.1 Performance

Table 2 reports the performance obtained with the three aforementioned program versions for each thread library. The first two programs were run on a uniprocessor machine whereas the last one was run on a dual-processor.

The basic version of the divide and conquer program makes heavy use of thread creations and synchronizations. As one may expect on a uniprocessor, the user-level MARCEL library is obviously the most efficient, while the LINUXTHREAD library exhibits poor performance, because kernel thread operations are much more inefficient than those related to user threads. It is interesting to note that the version using activations achieves good performance. The difference with the user-level version is due to the MARCEL lock acquire/release primitives that are a little more complex in the presence of activations.

With the version involving many I/O operations, things change significantly. The most noticeable result is the *huge* amount of time taken by the program with the user-level version. It is, however, not surprising: each time a user thread makes a blocking call, it blocks the entire UNIX process until a timer signal forces a preemption and schedules another thread (in this case, every $20ms$).

The activation version has the best execution time. The mixed MARCEL library does not behave as well because two underlying kernel threads are needed to handle the blocking calls properly. Thus, it introduces overhead due to additional synchronization and preemption costs.

When the program containing substantial computation is executed on a dual-processor machine, we observe that the activation version has approximately the same execution time as the MARCEL "mixed" and LinuxThread versions. It reveals that the activation version is perfectly able to exploit the underlying architecture by using two activations simultaneously within the application. The user-level version obviously performs poorly, because only one processor is used in this case.

4 Conclusion

This work augmented the design of activations, a new technique to handle thread support in an OS, then implemented and tested their use under Linux. We wrote a new version of the MARCEL thread library that utilizes activations while preserving the existing user interface, so that existing MARCEL programs still work with this new model. We have demonstrated that for applications using threads that make blocking system calls, performance of the new version of MARCEL on both single and dual processor platforms is superior to the best previous version of MARCEL and to kernel-level threads. Furthermore, since our new library is implemented in user space, we do not need to change the kernel to add new thread features, such as thread migration.

A two-level thread library based on activations seems to be a very attractive way to manage application threads. This work shows that this model is a valid one, in particular for application threads that utilize blocking system calls, which often happens within a communication library, for example.

References

1. T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.
2. Vincent Danjean, Raymond Namyst, and Robert Russell. Linux kernel activations to support multithreading. In *Proc. 18th IASTED International Conference on Applied Informatics (AI 2000)*, Innsbruck, Austria, February 2000. IASTED. To appear.
3. Xavier Leroy. The LinuxThreads library. http://pauillac.inria.fr/˜xleroy/linuxthreads.
4. R. Namyst and J.F. Mehaut. PM2: Parallel Multithreaded Machine. a computing environment for distributed architectures. In *ParCo'95 (PARallel COmputing)*, pages 279–285. Elsevier Science Publishers, Sep 1995.
5. R. Namyst and J.-F. Méhaut. MARCEL *: Une bibliothèque de processus légers*. Laboratoire d'Informatique Fondamentale de Lille, Lille, 1995.