LINUX KERNEL ACTIVATIONS TO SUPPORT MULTITHREADING

VINCENT DANJEAN
RAYMOND NAMYST
Laboratoire de l'Informatique du Parallélisme
École normale supérieure de Lyon
F-69364 Lyon Cedex 07, France
Email: Vincent.Danjean@ens-lyon.fr
Raymond.Namyst@ens-lyon.fr

ROBERT D. RUSSELL
Department of Computer Science
Kingsbury Hall
University of New Hampshire
Durham, New Hampshire 03824-3591, USA
Email: rdr@unh.edu

ABSTRACT

This paper describes a modification to the Linux operating system kernel that implements a mechanism called "scheduler activations" for supporting user-level multithreading. The key idea is to allow a user-level thread package to implement its own scheduling and to provide a mechanism whereby the kernel will notify this package whenever it (the kernel) makes a scheduling decision that effects one of the kernel-level threads utilized by the package. Based on this notification, the user-level scheduler decides which user-level threads to run. Tests show that a scheduler adapted to utilize this new mechanism gives performance comparable to the best performance of previous versions of the same scheduler and to kernel-level threads.

Keywords: threads, multiprocessors, activations, operating systems.

1 INTRODUCTION

Threads are the most popular paradigm for expressing concurrent activity in operating programs for shared memory multiprocessors. To date, thread packages utilize one of two fundamental implementation techniques: the package is implemented as a user-level library that is transparent to the operating system kernel, or the threads are implemented in the kernel itself as "lightweight processes". A hybrid of these two techniques is also found in some implementations: a user-level library that maps user-level threads onto kernel-level threads. Each of these methods has advantages and disadvantages.

1.1 User-level Threads

User-level threads are managed by the application, so they can be easily tailored to the particular requirements of the application. They are often very efficient, since they do not use additional kernel resources, and each process can easily have thousands of threads. The disadvantages are that because the operating system knows nothing about them, they cannot utilize all available system resources, especially multiprocessors. True parallel activity within one process is not possible. In addition, unless the interface to all blocking system calls is completely rewritten as part of the thread package, a thread making a blocking system call will block all threads belonging to the same process, even if some of them are ready to run.

1.2 Kernel-level Threads

Kernel-level threads are managed by the kernel, so they have access to all kernel resources, particularly multiple processors: several threads belonging to the same process can be simultaneously active, each with its own processor, so that true parallel activity occurs. Furthermore, when a thread makes a blocking system call, the kernel can allocate the processor to other threads belonging to the same process. The disadvantages are that kernel-level threads tend to incur much more overhead than user-level threads, since all thread scheduling and switching requires kernel intervention. Because they consume kernel resources, the number of available threads is usually severely limited on a systemwide basis, so that independent processes end up contending for threads as a scarce resource. Finally, because their properties are defined by the operating system, kernel-level threads can not be modified easily to accommodate any special needs of user applications.

1.3 Hybrid Threads

A hybrid approach, such as that taken by the MARCEL hybrid thread library, tries to obtain the advantages of both techniques by mapping user-level threads onto kernel threads. However, problems still exist. The mapping is usually fixed, such that one pool of user-level threads is always mapped onto the same single kernel-level thread. Although a user application can define several such pools, if one thread in a pool issues a blocking system call, no other threads in that pool can run. It can happen that every pool is blocked, even though each pool may contain threads that are ready to run. The cause of this degraded performance is that the kernel, which schedules the kernel-level threads, is unaware of the user-level threads mapped on top of them and therefore cannot remap the kernel-level thread when one of the user-level threads blocks. Another problem arises due to the interaction between threads in different pools. If, for example, a thread in one pool obtains a lock that is later needed by a thread in another pool, it would be prudent to ensure that the first thread is always allocated a processor. However, if the lock is implemented at the user level, for efficiency, then the kernel is unaware of that and may reallocate the processor assigned to the first thread. In this case, allocating the processor to the second thread may result in wasted processor cycles, since it is forced to spin-wait until the first thread releases the lock. Overcoming this problem by utilizing kernel-level locks greatly increases the overhead associated with locking, and significantly degrades performance in applications utilizing fine-grained locking.

2 SCHEDULER ACTIVATIONS

This paper describes a modification to the Linux operating system kernel that avoids many of the problems with hybrid threads. The key idea is to allow a hybrid thread package to implement its own user-level scheduling and to provide a mechanism whereby the kernel will notify this scheduler whenever it (the kernel) makes a scheduling decision that effects one of the kernel-level threads utilized in the hybrid mapping. Based on this notification, the user-level scheduler can adjust the mapping to avoid performance degradation.

This modification was first suggested by Anderson *et al.* [1] They describe a mechanism called *scheduler activations* in which a user-level thread scheduler effectively registers itself with the kernel in order to obtain *up-calls* from the kernel whenever it (the kernel) makes a scheduling decision effecting a kernel thread associated with that process. An up-call is essentially the opposite of a normal system *down-call.* In a down-call, a user-level process passes control and parameters to a fixed location in the kernel in order to accomplish a designated function. In an up-call, the kernel passes control and parameters to a fixed location in the user-level program in order to allow the user to take appropriate action based on the parameters. This typically involves management action relative to user-level threads. Up-calls would be made, for example, whenever a thread makes a blocking system call, whenever a blocked thread becomes unblocked, and whenever the kernel decides to preempt the processor from a running thread. Up-calls enable the user-level scheduler to regain control in situations where scheduling activity is required. On a system without up-calls, there is no efficient way a user-level scheduler can regain this control.

For whatever reason this mechanism has not been adopted by manufacturers when implementing their thread packages, even though the performance reported by Anderson *et al.* was very good. Their original prototype implementation was done for the DEC Firefly multiprocessor workstation, which is no longer functioning, and the sources were never released. Our implementation is therefore “from scratch” and is built upon the basic design concepts discussed by Anderson *et al.*, but differs in the details. We implemented our prototype as a set of modifications to the Linux 2.2.10 kernel. The basic up-call mechanism is general, but obviously requires a user-level thread package to take advantage of it. For that we modified a version of Marcel[2], a hybrid thread library developed as part of PM^2[3] (*Parallel Multithreaded Machine*). The resulting library requires no modifications to programs that utilize Marcel threads.

3 KERNEL MODIFICATIONS

3.1 Overview

We have chosen to implement activations as kernel threads. The kernel creates an activation by calling the function `do_fork()`, and then activates it by calling `upcall_new` to start execution of the user-level code. The first activation is distinguished as the `manager` activation because it is the repository in the kernel of common information that must be shared between all activations belonging to a single process. All other activations are kernel threads whose parent is the `manager` activation.

3.2 Modifications to kernel structures

Only one structure in the kernel had to be modified, the `task_struct` structure. A few fields were added to keep information about the state of the activation. The main fields are described here. For an exhaustive description of all the fields, refer to the sources or [4, 5].

In addition to information specific to each activation, some information needs to be shared between all the activations belonging to the same process. One possibility would have been to dynamically allocate and release a block of memory for this. But the space requirement is small, so we chose to add to the `task_struct` an additional structure, `struct_act_info`, and a pointer, act_info. The shared information is kept entirely in the struct_act_info structure of the manager activation, and the other activations access this information via the act_info pointer in their task_structs.. This structure handles data global to the process, such as the number of activations created, running, blocked, etc. and a spinlock to protect these data on SMP platforms.

3.3 Modifications to kernel functions

Only a few parts of the kernel had to be modified. Most of the modifications appear in the scheduler, which has to maintain information about the state of activations. This state is stored in the new fields added to the task_struct structure and allows the kernel to know what each activation is doing. The possible states of an activation are:

RUNNING the activation is running on a processor;

UPCALLING the activation is doing an upcall;

JUST BLOCKED the activation has just blocked in the kernel, but the upcall_block to notify the application has not been done yet;

BLOCKED the activation is blocked in the kernel and the upcall_block has been done;

JUST UNBLOCKED an activation that was blocked in the kernel has just unblocked, but the upcall_unblock to notify the application has not been done yet;

JUST PREEMPTED the activation has been preempted by the scheduler, but the upcall_preempt to notify the application has not been done yet;

PREEMPTED the activation has been preempted by the scheduler and the upcall_preempt has been done.

NEW the activation has just been created but the upcall_new to notify the application has not been done yet;

FREE the activation is not used any more (see 3.5 for more information);

ENDING the activation will be removed (the kernel thread will be ended).

An activation's state is modified by the scheduler when a process that uses activations is scheduled or unscheduled, and by the upcall code (see 3.4), when an upcall is done.

The kernel do_fork() and do_exit() functions had to be modified to initialize the new fields in the task_struct structure when a new activation is created, and to ensure that all the activations of a process end when the process ends. A process ends when one of its activations executes exit() without its state being set to ENDING (if the state were set to ENDING, it only means that the kernel does not need the activation anymore and that the kernel thread can be released).

Finally, the kernel's system call dispatcher was modified so that before each return to user mode the code that launches upcalls is executed if needed. This is achieved in the same manner as for signals. A variable need_upcall in the task_struct structure is set (by the scheduler, for example) each time an upcall is needed. When the kernel is returning to user mode after a system call or interrupt, it makes the necessary upcall if need_upcall is set.

3.4 The upcalls

Making an upcall is very similar to calling a signal handler. In fact, most of this code is derived from the code for signals. The differences are that we need to set up some parameters for the upcall function (*i.e.*, to put some values on the user stack), and sometimes we need to copy the user state (registers, *etc.*) of another activation. This happens when we need to give the application the state of an activation that has just unblocked. The main reason why we use upcalls instead of signals is their flexibility: it is easy to choose which stack to use and to pass as many parameters as are needed. If we had used signals only, then we would have needed to modify code in the kernel to allow us to change the stack, pass parameters and write information in user space, such as the state of another activation. In fact, we would have needed to write exactly what we wrote for the upcalls.

In our design, a single stack is used to make all upcalls. This stack is given to the kernel as a parameter to the act_new() system call when the application initializes the use of activations. Because we have only one stack, only one upcall can be in progress at any time, and it is necessary for the upcalled activation to call the act_resume() system call in order to inform the kernel when the upcall ends.

There are five kinds of upcalls:

upcall_new used when a new activation is launched;

upcall_preempt used to notify the application that an activation has been preempted;

upcall_block used to notify the application that an activation has made a blocking system call;

upcall_unblock used to notify the application that a blocked activation is unblocked;

upcall_restart used when one activation uses the system call `act_cntl(DO_UPCALL)` to request an upcall to another activation, for example to synchronize the activations.

3.5 Optimizations/Limitations

Whenever an activation blocks, a new activation must be started. Originally this caused the creation of a new kernel thread. To avoid this creation, we no longer delete kernel threads that become unneeded when a blocked activation unblocks. Instead, we put these threads into the `FREE` state and make them sleep on a wait queue. When we need a new activation, we first look for a free activation on this queue, and only if there are none then we create a new kernel thread.

Another optimization has also been made. Each time an activation blocks, the kernel originally needed to make an `upcall_new` to start a new activation and then an `upcall_block` to notify the application. We have merged these two upcalls together by giving the `upcall_new` an extra parameter that tells the application that another activation has blocked and which one it was. This eliminates the need for a separate `upcall_block`.

A limitation of our work is that signals are completely broken. For now, an application that uses activations cannot use signals. We do ensure correct behavior for the `KILL` signal, so that we can kill our applications, but the use of other signals can lead to unpredictable behavior. The main problem is due to the fact that an activation can be preempted at any time, and a pending signal in this activation will be lost if this activation is preempted and then deleted. However, we plan to restore correct signal management in a future version.

4 MARCEL MODIFICATIONS

The modifications needed to integrate activations into MARCEL consisted primarily of a few localized extensions. When it starts up, MARCEL calls `act_init()` which causes the kernel to initialize the activation mechanism for the process and to initialize the process's `task_struct` as the `manager` activation. Parameters to `act_init()` include the number of activations desired (between one and the number of physical processors), the stack to be used for upcalls, and a set of entry points which the kernel will use to make upcalls. Unless there is an error, control never returns from the `act_init()`. The kernel uses the number of activations requested in the `act_init()` to create that many `task_structs` as children of the `manager` `task_struct`, and then makes an `upcall_new` to each of these in turn to get them started. Each upcalled activation will call `act_resume()` to release the lock maintained by the kernel to ensure that a process has only one upcall in progress at a time, and will then start running a user-level thread.

MARCEL was also modified to keep a global pool of user-level threads, rather than a separate pool for each kernel-level thread as was the case in the hybrid version. We also had to ensure that the internal lock used by MARCEL to guarantee exclusive access to its internal data structures did not lead to a performance bottleneck when using activations. In particular, if the kernel preempts an activation that holds that lock, then that activation, rather than the activation receiving the `upcall_preempt`, is continued by MARCEL. Details of these and other modification issues can be found in [6].

5 PERFORMANCE

The new version of MARCEL on top of LINUX activations is completely operational, although we did not yet implement all the optimizations discussed in Section 3.5. 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 hybrid) as well as to the LINUXTHREAD library which uses native kernel-level threads [7]. The tests were performed 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.

We designed two distinct test programs to observe the behavior of MARCEL on top of LINUX activations. The first one (*Synchro test* in Table 1) aims at making heavy use of basic thread manipulation primitives. The corresponding 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 contains only 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.

The second test program (*I/O test* in Table 1) was designed to make extensive use of blocking UNIX I/O operations. For that, the corresponding program uses two threads that alternatively write to/read from a system *pipe*. Each thread executes this iteration fifty times, so that one hundred blocking I/O operations are involved (*i.e.*, when doing the `read()` system call).

Table 1 reports the performance obtained with these

Table 1: Performance of MARCEL using activations compared with other thread libraries (on a single processor machine)

Library	Synchro test	I/O test
MARCEL/user-level	0.308ms	1959.620ms
MARCEL/hybrid	0.435ms	0.761ms
MARCEL/activations	0.417ms	1.300ms
LINUXTHREAD	13.319ms	0.773ms

Table 2: Performance of MARCEL using activations compared with other thread libraries (on a dual-processor machine)

Library	Computation test
MARCEL/user-level	6932ms
MARCEL/hybrid	3807ms
MARCEL/activations	3551ms
LINUXTHREAD	3566ms

two test programs for each thread library.

Looking first at the results for the *Synchro test* program, the user-level MARCEL library is obviously the most efficient, while the LINUXTHREAD library exhibits the poorest performance. This is to be expected, because kernel thread operations are much more inefficient than those related to user threads, and the *Synchro test* program makes heavy use of thread creations and synchronizations. It is interesting to note that the library using activations achieves relatively good performance (4% better than the hybrid library). The 35% increase in time over the user-level library is due to the MARCEL lock acquire/release primitives that are a little more complex in the presence of activations.

With the test involving many I/O operations, things change significantly. The most noticeable result is the *huge* duration of the program run with the user-level version of MARCEL. 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 hybrid MARCEL library has the best execution time because we (manually) allocated two kernel threads for this program. The small (2%) difference with pure kernel threads is due to the optimized synchronization mechanisms which are very efficient in MARCEL. Note that this hybrid MARCEL library cannot allocate the "right" number of underlying kernel threads automatically. Moreover, if this number is not carefully set by the user, the performance drops considerably. Finally, we observe that the activation version does behave reasonably well, although the duration of the program is twice that of the kernel thread version. This is due to the fact that our current implementation of activations makes a large number of upcalls that could be reduced by appropriate optimizations (see Section 6). Actually, this *I/O test* program is the most aggressive "torture test" for our activation mechanism.

To evaluate the ability of the activation version to exploit multiprocessor architectures, we slightly modified the *Synchro test* program so as to augment the computation/synchronization ratio. Thus, a substantial speedup can be expected on multiprocessor architectures.

When executing this latter test program on a dual-processor machine, we observe (Table 2) that the activation version has approximately the same execution time as the MARCEL hybrid 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. Obviously, the user-level version performs poorly because only one processor is used in this case.

6 ONGOING WORK

Due to the way the MARCEL thread library uses activations, we see a number of improvements that can be made in order to simplify the code and increase performance. The first observation is that we can eliminate the `upcall_block` and the corresponding extra parameter to `upcall_new` because they are not used anywhere in the thread library. Whenever a thread blocks in the kernel, the kernel keeps the number of running activations for that process constant by starting another activation and issuing an `upcall_new` to it. Until the blocked thread unblocks, the thread library considers its activation to be running and does not need to know that the thread is, in fact, blocked in the kernel. When the blocked thread does unblock, the kernel releases its thread and notifies the thread library by making an `upcall_unblock` to one of the unblocked activations in order to reschedule the user-level thread.

Another observation concerns the locks. The current thread library uses `upcall_preempt` because it needs to know when an activation holding the `marcel_lock` is preempted in order to relaunch this thread as soon as possible. But on a uniprocessor, we never need to do anything special to take care of the activation holding the `marcel_lock` because that is always the only running activation. On a multiprocessor, special action is useful only if there are processes other than ours running at the same time (because otherwise all processors would be running our unblocked activations). Therefore, we believe we can eliminate the `upcall_preempt` and the lock management. They would be useful on multiprocessors when many other processes are running, but without them we will remove considerable overhead from the thread library.

Furthermore, we believe that the kernel can use the stack of the unblocked activation when making an `upcall_unblock`, and that the kernel can have as many stacks as processors for the `upcall_new`. These simplifications will remove the present limitation of having only one outstanding upcall at a time, and will make it easier to add automatic preemption (with a timer) and to provide signal handling to user-level threads.

7 CONCLUSION

This paper describes modifications to the LINUX kernel that introduce a mechanism called *scheduler activations* in order to provide support for user-level multithreading. We also modified the MARCEL thread library to utilize these activations, and then compared the performance of test programs using the new library against the same test programs using several existing libraries.

Compared to the original user-level threads version of MARCEL, the new version introduces a small additional overhead in applications where the original version experienced no problems. However, in applications where user-level threads invoke blocking system calls, the performance improvement of the new version is enormous because blocking system calls issued by one user-level thread no longer block any other user-level thread.

The new version shows a slight (4%) improvement in performance over the hybrid version of MARCEL in situations involving no blocking I/O, but tests that utilize blocking I/O take about twice as long with the new version primarily because of the large number of upcalls involved. However, we have discussed a number of unimplemented optimizations that we believe will improve the new version considerably in these situations. In any case, the new version has more potential than the hybrid version because it is no longer necessary to "pool" groups of user-level threads for fixed mappings onto the underlying kernel threads. This is important because the hybrid version cannot automatically allocate the "right" number of kernel threads and performance is very sensitive to this number. This gives the new version of MARCEL a substantial practical advantage over the hybrid version.

Compared to the LINUXTHREAD library the new version of MARCEL exhibits a large improvement in performance on tests that involve only user-level thread manipulations, such as creation, synchronization, *etc.* This is true of all previous versions of MARCEL as well, so nothing new has been gained here. However, the new version shows a factor of 2 disadvantage compared to kernel-level threads in tests involving blocking I/O. As discussed above, we believe further optimizations will reduce this considerably.

In summary, the version of MARCEL based on activations retains all the advantages of a user-level version, such as the ability to create a huge number of user-level threads, and to switch quickly between user-level threads, while also retaining the advantages of kernel-level threads, such as the ability to handle blocking system calls. Although present performance is not quite up to that of the hybrid library in all situations, we have strong reason to believe this can be corrected quickly. In any case, the version based on activations is more powerful than the hybrid version because it does not require manual intervention to obtain the correct number of underlying kernel threads.

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] R. Namyst and J-F. Méhaut. MARCEL *: Une bibliothèque de processus lègers.* Laboratoire d'Informatique Fondamentale de Lille, Lille, 1995.

[3] R. Namyst and J-F. Méhaut. PM2: Parallel Multithreaded Machine. a computing environment for distributed architectures. In *ParCo'95 (PARallel COmputing)*, pages 279–285. Elsevier Science Publishers, Sep 1995.

[4] V. Danjean. A New Interface for Scheduler Activation Support in the Linux Kernel. http://www.ens-lyon.fr/~vdanjean/activations/, 1999.

[5] V. Danjean. Internals of Activation Support in the Linux Kernel. http://www.ens-lyon.fr/~vdanjean/activations/, 1999.

[6] Vincent Danjean, Raymond Namyst, and Robert Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4rd Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, Lect. Notes in Comp. Science, Cancun, Mexico, May 2000. Held in conjunction with IPPS/SPDP 2000. IEEE TCPP and ACM SIGARCH, Springer-Verlag. To appear.

[7] Xavier Leroy. The LinuxThreads library. http://pauillac.inria.fr/~xleroy/linuxthreads.