Avoiding Blocking System Calls in a User-Level Threads Scheduler for Shared Memory Multiprocessors

Andrew Borg
andborg@orbit.net.mt
University of Malta

June 26, 2001

Abstract

SMP machines are frequently used to perform heavily parallel computations. The concepts of multithreading have proved suitable for exploiting SMP architectures. In general, application developers use a thread library to write such a program. This library schedules threads itself or relies on the operating system kernel to do so. However, both of these approaches pose a number of problems. This dissertation describes the extension of two user-level thread schedulers, one for uniprocessors and one for SMPs. This will enable the execution of blocking system calls without blocking the scheduler kernel. In order to do this we make use of an OS extension called scheduler activations. The usefulness of avoiding blocking system calls is apparent in server applications which need to handle multiple clients simultaneously. A web server that is capable of dispensing static pages and which implements the HTTP1.0 protocol is also implemented to demonstrate the effectiveness of this approach.

1 Introduction

In older UNIX based operating systems, support for concurrent programming was unsatisfactory. This made the design and execution of intrinsically parallel applications difficult. The solution lay in a new structuring by which these programs could be defined as a series of independent sequential threads of execution that communicate and cooperate at a number of discrete points. Multithreading is the vehicle for concurrency in many approaches to parallel programming. The structural independence of each thread in an application lends naturally to the idea of using more than one processing element to execute the program.

Traditionally, threads could be divided into user threads and kernel threads. Efficiency is the main advantage of user-level thread libraries as the scheduler kernel operates in user space. Kernel threads are less efficient as they require the OS kernel to provide the scheduling mechanisms. However while user threads have no support from the kernel, kernel threads benefit from this property. In this dissertation we shall be making use of a system called scheduler activations[1] to provide kernel level support for a user-level thread library. In this way, user threads maintain their efficiency as the scheduler still operates in user space. However, the kernel is also able to provide information on its scheduling decisions and provide support to optimise thread scheduling at the user level.

2 Background

2.1 From Processes to Kernel Threads

The notion of a thread, as a sequential flow of control, dates back at least to 1965, the Berkeley Timesharing System being the first instance. At the time they were not called threads, but were defined as processes by Dijkstra[4]. The early 1970s saw the birth of the UNIX operating system.

The UNIX notion of a process became a sequential thread of control *plus* a virtual address space Thus, processes in the UNIX sense are quite heavyweight machines.

Processes were also designed for multi-programming in a uniprocessor environment, making them suited for coarse-grain parallelism but not for general purpose parallel programming. This led to the 'invention' of threads which were nothing else than old-style processes that shared the address space of a single UNIX process. They were also were called 'lightweight processes', by way of contrast with 'heavyweight' UNIX processes. Lightweight processes (LWPs) are often referred to as kernel threads because it is the kernel that is responsible for all creation, destruction, synchronisation and scheduling activities. This has the advantage that since the kernel is aware of these threads, all scheduling policies that hold true to processes can be applied to LWPs, including true timeslicing.

2.2 From Kernel Threads to User Threads

The major disadvantage of kernel threads is that they are limited to the functionality provided by the system's kernel. For example the application developer is bound by the scheduling constructs provided by the kernel thread library. Some applications require nonstandard properties for their threads that are not offered by the kernel. Kernel threads, though lightweight when compared to processes, are still heavily dependant on kernel resources.

User threads provide an alternative to kernel threads and instead operate on top of kernel threads. Figure 1 shows this arrangement. The user threads are multiplexed across the kernel threads which are in turn multiplexed across the CPUs. In an application that utilizes user threads, the threads of the application are managed by the application itself. In this way, functionality and scheduling policy can be chosen according to the application. These user threads are much more efficient than kernel threads in carrying out operations such as context switching, since no kernel intervention is necessary to manipulate threads. The performance of kernel threads, although an order of magnitude better than that of traditional processes, has been typically an order of magnitude worse than the best-case performance of user-level threads[1].

2.3 The Problem of User Threads - Poor system Integration

As a consequence of lack of kernel support in existing operating systems, it is often difficult to implement user-level threads that have the same level of integration with system services as is available with kernel threads. The kernel, in turn, is not aware of multiple user threads created on top of kernel threads. This leads us to the two related characteristics of kernel threads that cause difficulty:

- Kernel threads are scheduled obliviously with respect to user-level threads state.
- Kernel threads block and resume without notification to the user level.

The first problem is apparent when scheduling user threads that have different priority levels. The kernel is unable to distinguish between these priority levels and, therefore, a user thread with high priority might be preempted in order to execute a user thread with a lower priority. A worst-case scenario would be the kernel preempting a kernel thread that is running a user thread holding a lock. It might then give processor time to another kernel thread that has a user thread spinning and waiting for the lock to be released.

The second problem concerns our main interest and is how to deal with the blocking and resumption of kernel threads in a user-level threads library.

2.4 Blocking System Calls

When an application requires a service from the kernel, it does this by means of a system call. System calls provide the interface between user-level applications and the kernel. They are often used when the application requires the services of the underlying system hardware, such as the internal clock or an I/O device. The operating system services the request and returns a result depending on the definition of the system call.

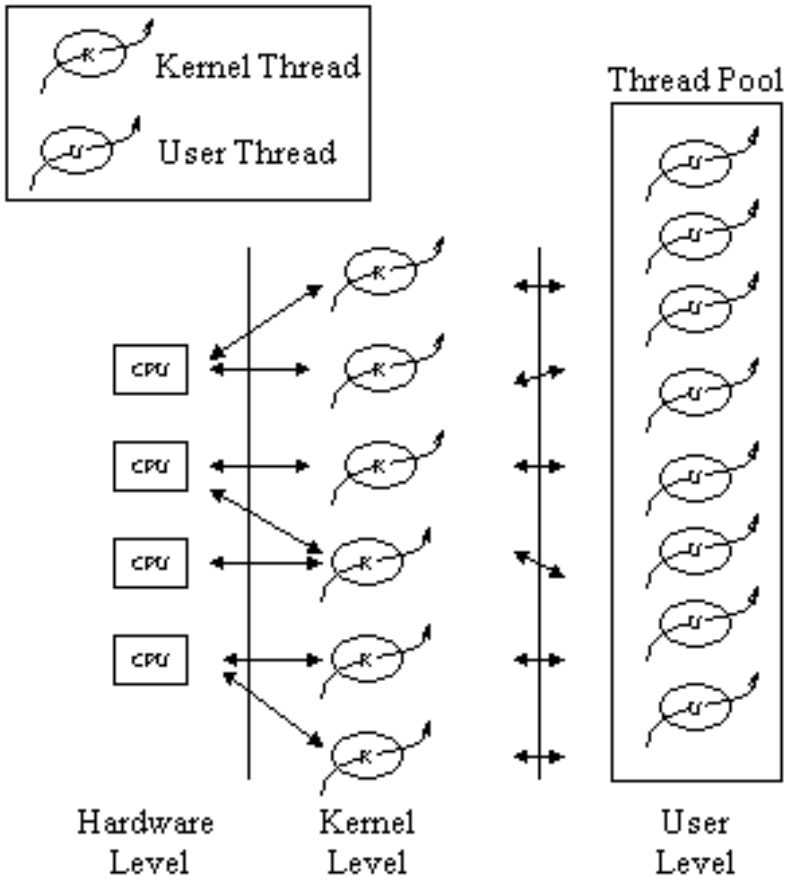

Figure 1: User and Kernel Threads Allocation

It is, however, sometimes impossible for the kernel to immediately service a request because the underlying hardware is not ready to provide the required service. For example if a read() system call is made from an input device such as a disk, the device may still be busy reading the data from it's medium. In this case, the kernel sets the kernel thread to sleep and services other threads. It is only when the request is finally fulfilled that the thread is unblocked and placed back onto the run queue.

In any application that uses a system where the number of kernel threads is the same as the number of processors (as is true for the user-level thread libraries we shall be using), we end up in a situation where we have less running kernel threads that we have available processors. This under-utilization of resources means that we could very well have user threads that are waiting to be run and processors that are idling whilst waiting for threads to unblock. This is a condition that arises simply because there is no communication between the user-level scheduler and the OS kernel.

3 Scheduler activations

Scheduler activations were originally proposed by Anderson *et al*[1] at the University of Washington. Its authors implemented this mechanism on top of the FastThreads library on the Topaz system. This system unfortunately is no longer running and the source was never released. In this section we shall be describing in detail the operation of Anderson's scheduler activations, in particular how the system behaves when a blocking system call is made by the application.

3.1 The Idea Behind Scheduler Activations

Scheduler activations enable the kernel to notify an application whenever it makes a scheduling decision affecting one of its threads. Anderson[1] coined the term 'Scheduler Activation' because each event in the kernel causes the user-level thread scheduler to reconsider its scheduling decision of which threads to run on which processors. This mechanism is implemented by introducing a type of system call called an *upcall*. While a traditional system call from an application to the application can be termed a downcall, an upcall is a call from the kernel to the application. In order to make these upcalls, the kernel makes use of a scheduler activation. A scheduler activation is an execution context in exactly the same way that a normal kernel thread is. In fact, implementations of activations use the operating system's native kernel threads and simply add the functionality of upcalls. When an application uses a classical kernel thread, it creates that thread itself and designates a function for the operating system to execute. The opposite happens with scheduler activations. The operating system decides when an activation is needed. It then creates it and begins executing a specific user function.

3.2 Activations and Blocking System Calls

Our main interest lies in how scheduler activations deal with blocking system calls. What follows is a simple example of an implementation of activations that has the bare minimum to deal with blocking system calls. The implementation we use in our thread libraries is based on the same idea but uses a different set of upcalls and downcalls.

First of all, we shall define three required upcalls:

1. **upcall new** This upcall is used to notify the application that a new activation has been created. The application can then use this activation to run the code it requires.

2. **upcall block** When an activation blocks, the kernel uses *another* activation to make this upcall in order to notify the application that one of its activations has blocked.

3. **upcall unblock** This upcall is made to report to the application that one of its activations has become unblocked. The application can then resume the user thread that was running on that activation.

Figure 2 illustrates what happens when on a dual processor machine, an application using activations makes a blocking I/O request.

Time T1: The kernel allocates two processors to the application. Two new activations are launched with the new() upcall. The user-level threads library selects two threads from the pool and begins to run them.

Time T2: Activation (A) makes a blocking system call (such as an I/O request). A new activation is created using the new() upcall. An upcall block is also made to notify the user level that one of its activations has blocked. This allows it to take appropriate action such as removing the thread from its run queue. The threads library then chooses another thread from the thread pool and uses the new activation (C) to run it.

Time T3: The activation (A) finally unblocks (for example on completion of the I/O request), and the application receives notification from the kernel by means of an unblock upcall. This upcall could be made either through activation (B) or (C). At this point, the activation performing the upcall can choose to continue its thread or to immediately resume the thread that was blocked. In any case, the extra activation is discarded. In this way, the number of *active* activations is always equal to the number of available processors. This removes any unnecessary context switching in the kernel.

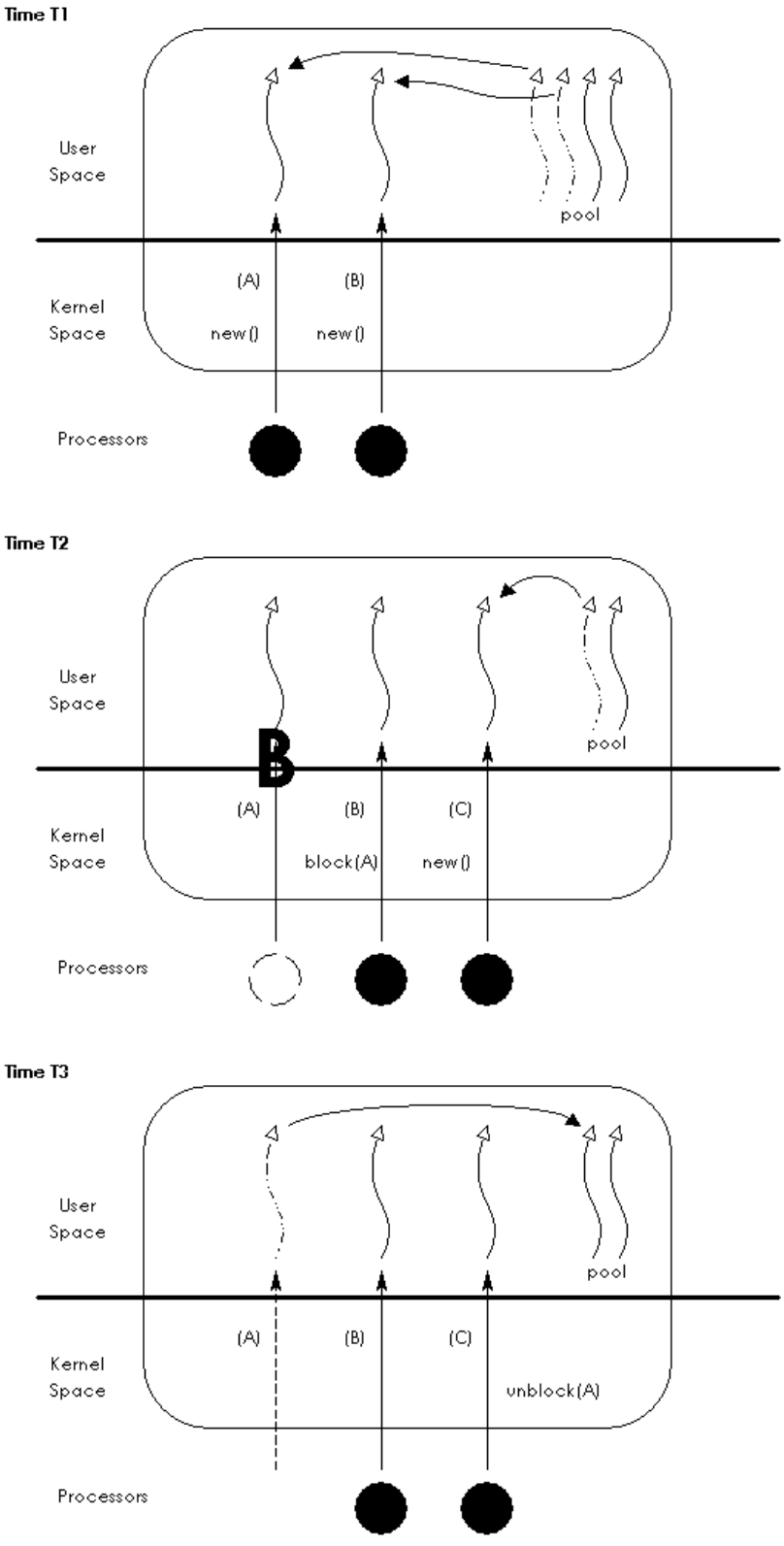

Figure 2: A blocking system call with activations

3.3 Scheduler Activations with Polled Unblocking

In Anderson's model for scheduler activations, all interaction between the OS kernel and user-level schedulers is done synchronously. Synchronisation between the kernel and user-level schedulers necessarily involves horizontal and vertical switching. Hence with a synchronous interface, unnecessary switching is introduced into the system. A solution proposed by Inohara and Masuda[6] is to use a shared memory area to provide an asynchronous interface between the kernel and user space. Using this idea, Danjean creates a new model for scheduler activations where a blocking system call automatically creates a new activation in exactly the same way that Anderson's model did. However, when an activation unblocks, no upcall is made. Instead an integer value in shared memory is incremented. The application then polls this value and restarts unblocked activations when it deems fit.

4 Uniprocessor Implementation

We shall be implementing two versions of our uniprocessor thread library with support for scheduler activations. The first uses the original model as proposed by Anderson. The second uses the model with polled unblocking. Support for each version of scheduler activations comes from a patch written for the Linux kernel by Danjean[2, 3] at Lyon.

4.1 The Uniprocessor Scheduler

The uniprocessor scheduler used is **smash**. This scheduler was developed at the University of Malta by Debattista[5]. The uniprocessor **smash** is based on the MESH[7] scheduler. **smash** is a non-preemptive based user-level thread scheduler that borrows many of the ideas used in MESH to increase performance and speedup. It strips off the external communication interface. Besides providing an API for thread management, **smash** also provides an API for CSP constructs. The version we shall be using for our implementation is the simple circular run queue without a dummy-head.

Integrating scheduler activations with our thread scheduler basically involves:

- modifying the thread library to deal with special conditions that arise because of the use of activations.
- defining structures to store information on activations.
- writing code for each of the upcalls mentioned above.

4.2 Scheduler Activations with Synchronous Unblocking

The scheduler activations patch used for synchronous is built around the original model of scheduler activations as proposed by Anderson and described in section 3.1. This version of the scheduler activations patch is not as stable as the second and suffers from a number of bugs. It is however interesting in that it allows us to study the nature of race conditions that arise when using this model.

4.3 Key Implementation Considerations - Synchronous Unblocking

Race Conditions As a result of the asynchronous nature of this implementation, we have to be very careful in recognizing and solving potential race conditions. For example when an activation unblocks, it is not always safe to simply enqueue the unblocked user-thread onto the run queue. An unblock upcall can occur at any time. Therefore if we are modifying our run queue and an unblock upcall occurs, a race condition will arise were we have two threads of execution modifying the same data structure. For this reason, when an activation unblocks, we place the unblocked thread into a temporary linked-list. These unblocked threads are placed onto the scheduler run queue only when

we can guarantee that no other operation on the run queue data structure is in operation. This is done using an atomic SWAP operation (see [9]).

4.4 Scheduler Activations with Polled Unblocking

The scheduler activations patch used for the second implementation is built around the hybrid model of scheduler activations as implemented by Danjean and described in section 3.3. This version of the scheduler activations patch is far more stable than the first. It operates on the Linux 2.2.17 kernel though a version for the 2.4 kernel is expected to be released in the near future. Unfortunately since this implementation is very recent, there are no references that can be made to any publications or manuals. In fact, it was necessary to study the source of the patch in order to extract the API.

4.5 Key Implementation Considerations - Polled Unblocking

Race Conditions Since we are implementing our non-preemptive scheduler on a uniprocessor and no preemption is possible from unblocked activations, we no longer need to worry about race conditions. Therefore we are able to do away with the atomic SWAP operation and the temporary linked list of unblocked threads. Any threads that unblock can be safely placed onto the run queue by standard instructions. Integration with our scheduler is also much simpler and the flow of execution is much easier to comprehend. However, complications come in some lacking properties of the API.

Storing Activation Information The new API however, unlike the previous version, does not provide us with an integer to identify an activation. We need this to save a reference to the user thread that was running on it. This is required so that when that activation unblocks, we are able to place the unblocked user thread back onto the run queue. We must therefore find some other method to identify activations. Several solutions exist to this problem. The solution we use is based on the solution used by Glibc[11] and also in the original SMP thread scheduler which we shall be using in the next section. This system works by making a system call called modify_ldt() to set up a segment for each activation. We store information in the segment relative to our kernel thread. In our case we store a pointer to an activation structure. This structure stores:

- An integer value by which we identify the activation.
- A pointer to the user thread that was running on that activation before it blocked.

A further problem arises in deciding how to store these structures. An obvious solution is to use an array. However, this time the API of the patch does not allow us to set a limit to the maximum number of activations that can be blocked in the kernel. This means that we still need to create these structures dynamically so an array implementation is not possible. Instead of creating one structure at a time we create a whole block of structures at one go in an array. We then store these arrays in a linked list. We also keep a linked list to reference which structures are free and which are used within these blocks. When the number of free structures becomes zero, we create a new block with just one malloc() operation. Using our linked list of arrays, if we have N structures in total the number of blocks is K (the blocksize is therefore N/K), the order is O(K). With a normal linked list implementation this would have been O(N).

5 SMP Implementation

The support for scheduler activations for our SMP scheduler comes from the second kernel patch described in the previous section.

5.1 A user-level scheduler for SMPs

The version of **smash** that we shall be using is the shared run queue with coarse grain locking. The number of processors and kernel threads are always equal and each kernel thread is bound to a single processor. This model fits exactly that used in scheduler activations where there is always one active activation per processor. However blocking system calls severely degrade performance for reasons discussed in section 2.4.

5.2 Key Implementation Considerations

Shared Data Structures Many of the structures needed to extend the SMP scheduler to support scheduler activations have already been implemented for the uniprocessor scheduler. Since we are now working in a multiprocessor environment, more than one process may be attempting to modify some shared data structure such as the list of unblocked threads. If this occurs, that shared data structure may be corrupted. One could consider such a structure to be a resource that can only be used by one task at a time. Access control for these resources is provided by spin locks[8].

Sleeping and Waking Up The original version of the scheduler without scheduler activations made use of semaphores for the idling system. Since the operations on semaphores are blocking system calls, this is no longer possible. Instead the scheduler activations API provides us with three system calls which when used together produce the desired result. Two of the system calls are used together to put activations to sleep, the other to wake them up. This is necessary because, unlike the operations on semaphores, the sleep and wake operations could lead to the lost-wake-up problem [10] where an activation may be erroneously set to sleep. This problem is solved using a system of busy waiting and the two system calls.

Race Conditions The implementation of this scheduler activations patch has an extra optimization in that unblocked activations can be restarted automatically if a processor is idle. This leads to a number of potential race conditions. For example, if an idle processor is about to be woken up to execute a user thread and is instead unblocked automatically, a race condition may occur and shared data structures may be corrupted. In order to ensure correct execution it is necessary that all such race hazards are identified and resolved.

6 Web Server

In order to demonstrate and test the schedulers, we develop a web server, **ActServ**, that is capable of dispensing static pages. The web server implements the HTTP 1.0 protocol as described in RFC1945. Rather than forking a new process for every request, a user-level thread is used to service a client. With scheduler activations we also have the advantage that blocking system calls do not block the web server.

6.1 Results

Figure 3 shows the results obtained for **ActServ** and those obtained from the Apache Web Server. Apache is configured beforehand to create as many processes as required to handle requests. This is correct as we do not limit the number of activations that can be created by our web server. In order to carry out the test we use a benchmark program that creates (c) processes, each making 500 or 1000 sequential requests. Results are on average 25% better than those for Apache. We do not consider the uniprocessor thread scheduler without activations as no concurrency can be achieved at all since the blocking system calls block the whole application.

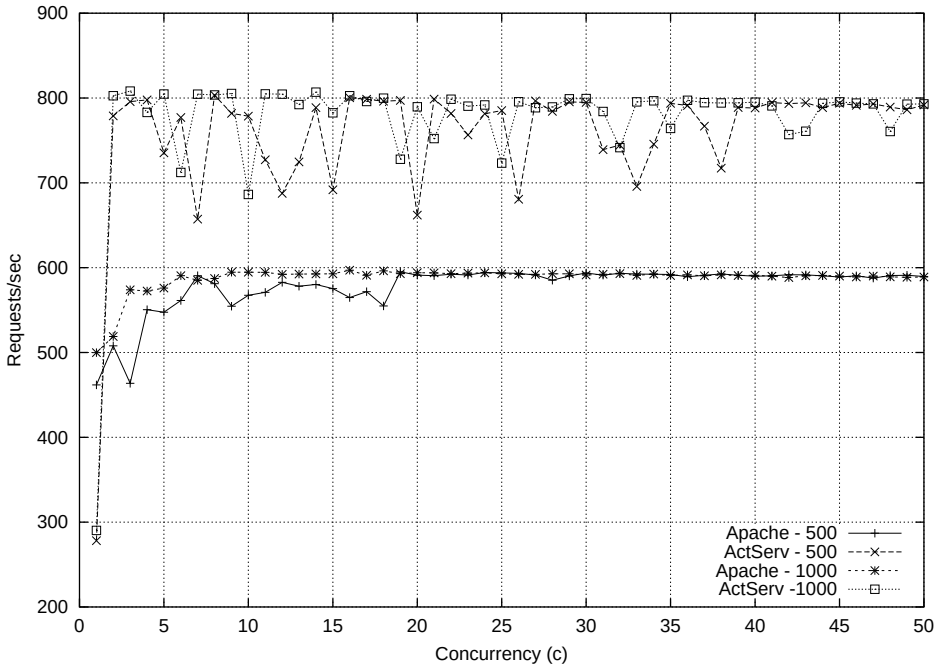

Figure 3: Uniprocessor web server results

7 Conclusion

Scheduler activations have been fully integrated into two user-level thread schedulers, one for uniprocessors and one for SMPs. Results from synthetic tests and the web server show a marked improvement in applications that use blocking system calls. The overhead introduced in order to provide kernel support for user threads is easily compensated for by avoiding blocking system calls blocking the underlying kernel thread.

References

[1] T. Anderson, B. Bershad, E. Lazowska and H. Levy,*Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* Proc. 13th ACM SOSP, October 1991

[2] Vincent Danjean, Raymond Namyst, Robert D. Russell,*Integrating Kernel Activations in a Multithreaded Runtime System on top of Linux .* cole Normale Suprieure de Lyon, March 2000.

[3] Vincent Danjean*Extending the LINUX Kernel with Activations for Better Support of Multithreaded Programs and Integration in PM2.* cole Normale Suprieure de Lyon, Internship made at the University of New Hampshire, September 1999.

[4] E. W. Dijkstra,*Cooperating Sequential Processes.* 'Programming Languages', Genuys, F. (ed.), Academic Press, 1965.

[5] K. Debattista,*High Performance Thread Scheduling on Shared Memory Multiprocessors.* Thesis for the Degree of Masters of Science, University of Malta, January 2001

[6] S. Inohara, T. Masuda,*A Framework for Minimizing Thread Management Overhead Based on Asynchronous Cooperation between User and Kernel Schedulers.* Technical Report Department of Information Science, Faculty of Science, University of Tokyo, January 1994

[7] M. Boosten, R.W. Dobinson and P.D.V van der Stok,*Fine Grain Parallel Processing on Commodity Platforms.* Fine Grain Parallel Processing on Commodity Platforms. IOS Press, 1999.

[8] U. Vahalia,*UNIX Internals.* Prentice Hall, 1996.

[9] C. Schimmel,*UNIX Systems for Modern Architectures.* Addison Wesley, 1994.

[10] A. M. Lister R.D. Eager,*Fundamentals of Operating Systems.* MacMillan Computer Science Series, 1993.

[11] Glibc,*http://www.gnu.org/software/libc/libc.html*

[12] The Apache Foundation,*http://www.apache.org*