# Sleeping Threads: A Kernel Mechanism for Support of efficient User Level Threads

Christoph Koppe

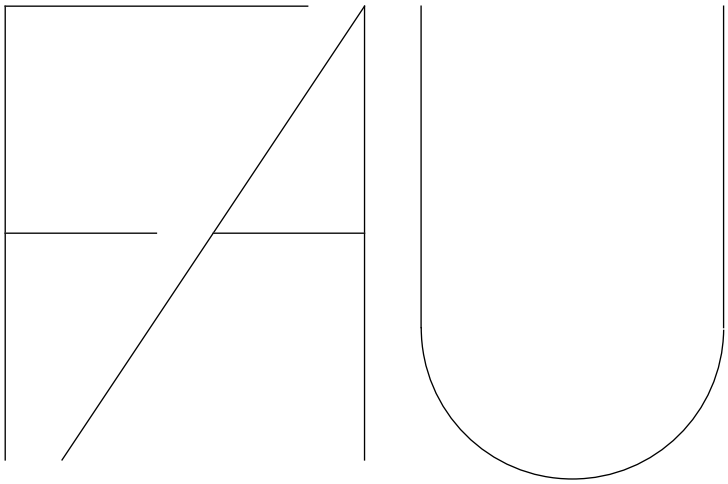
August 1995 TR-I4-07-95

Technical Report

Computer
Science Department

Operating Systems - IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



This paper is accepted for the Seventh IASTED-ISMM International Conference on Parallel and Distributed Computing Systems (PDCS'95), Washington D.C., October 18-21, 1995. It will be published in the conference proceedings.

# Sleeping Threads: A Kernel Mechanism for Support of Efficient User Level Threads*

Christoph Koppe

University of Erlangen-Nürnberg
Computer Science Department - Operating Systems - IMMD IV
Martensstraße. 1, 91058 Erlangen, Germany
email: chkoppe@informatik.uni-erlangen.de

**Abstract.** The rapid development of modern parallel computers in the last years allows - from hardware point of view - a finer granularity of parallel algorithms. Besides faster low latency networks for message passing based architectures the development of shared memory systems was enforced. But particularly for these systems there is a lack of sophisticated system software. The very efficient communication via shared memory can't be used in a comfortable way. The threads of the operating system (kernel threads) suffer from high system overhead and can't be used for applications with fine grain parallelism. In most cases they only build the base for an efficient thread management on user level. On the other side efficient user level threads suffer from insufficient system integration which causes problems and low performance. However, the insufficient system integration of the user level threads is not a problem of the thread libraries themselves but of the operating systems. Most of them allow multiple kernel threads per address space as the only support for parallel programming. A well integrated user level scheduling needs more support and for example information about the thread states in kernel. In this paper we describe a method to achieve better system integration without many changes in the operating system kernel. This ensures a unproblematic port of this mechanism to other systems.

**Keywords:** Two level scheduling, high performance computing, NUMA architectures, operating systems.

## 1 Introduction

The increasing appearance of parallel architectures in the field of high performance computing causes a new paradigm in software development. For conventional super computing systems - in the sense of vector computers - a rewriting of applications was not necessary in general. Old FORTRAN code could be ported with minor changes. The programmer only had to ensure, that the algorithms make use of the vector units. On the other hand porting to parallel architectures requires fundamental changes in code. Whole applications have to be rewritten using new programming models (message passing or shared memory based). Message passing based models became the new programming paradigm for high performance computing in the last years, while only a few shared memory based system were present in this field. One, but not the only, reason for this was the bottleneck in accessing shared memory via a common memory bus. This bottleneck prevents systems with processor numbers comparable to those of message passing architectures. With the appearance of NUMA architectures this problem is solved. Programming models for shared memory systems will be of more importance in the future.

The concept of light weight processes or threads was the successful attempt in using shared memory based parallel systems for fine granular parallel algorithms. But the kernel threads still suffer from a high overhead for high performance computing applications. The solution is an efficient scheduling at user level, using the kernel threads as virtual processors. But this leads to some other problems because of insufficient system integration of the user level threads [1][2]. Some of these problems are:

- It is not possible to signal a specific user level thread.
- If a user level thread is blocked in kernel the underlying kernel thread (virtual processor) is also blocked and inside the application parallelism is lost.
- The user level scheduling strategy can interfere with kernel scheduling (two level scheduling).

We think that the main part of these and other problems can be solved by communication between kernel and user level thread library with only a few modifications and extensions to the kernel itself. In this case the communication is an extended bidirectional communication, not only in terms of system calls but also initiated by the kernel. This paper describes the *Sleeping Threads*, a mechanism which uses communication between kernel and user level to address the problems of kernel blocked user level threads and interferences between user level and kernel scheduling. The *Sleeping Threads* mechanism basically implements the functionality of parking spare threads in kernel and shifting

* This work was supported by the Bavarian Foundation for Scientific High Performance Computing (FORTWIHR) and Convex Computer Cooperation, Richardson, Texas.

all scheduling decisions from kernel to user level. The problems mentioned above must be solved by user level strategies based on the new mechanism.

## 2 Communication between kernel and user level

There are two extreme ways to implement a new user level functionality which needs kernel modifications: Integrate as much as possible in kernel, offering a comprehensive interface to the user level, or integrate as few as necessary as a base for a user level implementation. The first attempt has the advantage that an integration into the kernel simplifies the implementation. All kernel data can be accessed and the system state is mainly controlled by the kernel as the only instance. Limits for a kernel integration are given by efficiency and kernel overloading. The disadvantage of this attempt is the necessity of extensive modifications in kernel. Easy and quick modifications or extensions are not possible. The second attempt can lead to a more complex implementation. This is because of the distribution of status information between two instances. But the second attempt also has important advantages[1]:

- Easy separation of mechanism and strategy (strategies are realized in libraries).
- Easy exchange of strategy modules.
- Parts realised in user level have not to be checked for system integrity.
- Good portability to other systems.
- Easy to achieve uniform interfaces over system boundaries.

For obvious reasons modifications and extensions of the kernel itself can be minimized by using a bidirectional communication between kernel and user level. Communication allows integration of functionality mainly at user level. In the other case extensive kernel modifications are unavoidable.

These reflections do not only apply to user level scheduling but also to other problems like signalling user level threads. Even communication between application and thread library can make sense. An example for this is the necessity of knowledge about future scheduling decisions. This allows a more efficient use of the cache resources. Especially for automatically generated parallel code this can be used to prefetch cachelines in time.

The long term aim of this work is to examine the effects of a communication between kernel, thread library and application concerning efficiency and capability of the programming model “threads”.

1. This arguments are known from the discussion about monolithic and minimal kernels with external servers.

## 3 Sleeping Threads

### 3.1 A problem of user level threads

The following scenario demonstrates the problem user level threads blocking in kernel. A thread library binds a number of user level threads to a smaller number of kernel threads (virtual processors) which matches the number of available processors. All user level threads are runnable. If one of the user level threads executes a blocking system call, the bound kernel thread and therefore the virtual processor blocks. Inside the application a virtual processor is lost and a physical one is unused, even if there are runnable user level threads. The state of the user level thread isn’t changed because the thread library can’t detect a blocking in kernel. This problem appears frequently, if user level threads use blocking system calls. In many cases this can be avoided by the programmer. But the use of threads is certainly easier, if the programmer doesn’t have to consider things like this. The blocking of user level threads is not avoidable in any case. At least a blocking caused by page faults can happen in any application. To avoid such a loss of parallelism inside an application, more than one virtual processor must be bound to a physical one. This results in loosing the advantages of a pure user level scheduling, because the kernel has to schedule virtual on physical processors.

An application can run into a deadlock situation if the same number of user level threads block as virtual processors are available. At least this problem has to be solved to allow blocking system calls for user level threads. One solution is to bind each user level thread to it’s own kernel thread. This, of course, makes no sense because the scheduling is done completely in kernel as a consequence. A more reasonable solution is used by SUN for the threads of the Solaris System [3]. If the kernel, via a timeout, detects all virtual processors (the Solaris „lightweight processes”) as blocked, it sends a special signal to the thread library. A thread is released and resolves the situation by creating an additional kernel thread. Of course, this is a mechanism to resolve, not to prevent such a situation.

### 3.2 The basic mechanism

The features of the *Sleeping Threads* are more powerful. A thread library can park spare threads in kernel. In the case of a blocking in kernel the mechanism can switch very fast to one of the spare threads. The thread library must implement strategies which have to ensure the availability of spare threads. These strategies can be application dependent. All scheduling decisions, even the rescheduling of in-kernel blocked threads, are shifted into the thread library. In conjunction with other mechanisms, to bind an application exclusively to a set of processors, interfering kernel scheduling can be avoided completely. This can be a job or gang scheduling strategy [4][5], a processor reservation mechanism like the MACH CPU Server [6] or some other mechanisms for processor scheduling between applications integrated in the *Sleeping Threads*.

To describe the *Sleeping Threads* mechanism we need to introduce some new thread states. To implement the *Sleeping Threads* we need three additional kernel thread states: *parked (sleeping) Sleeping Thread* (**SS**) for a parked spare thread, *blocked Sleeping Thread* (**SB**) and *deblocked Sleeping Thread* (**SD**) for a blocked or already deblocked *Sleeping Thread*. For a user level thread we need two additional states: *blocked in kernel* (**BK**) and *deblocked in kernel* (**DK**). In the following figures we also need the state *runnable* (**R**) for a runnable user level thread. We assume that user level threads are always runnable. The case of suspended user level threads is not of interest for the described mechanism. The notation is shown in figure 3.1.

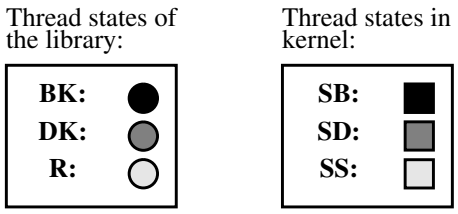



**Fig. 3.1: Symbol definitions**

We want to schedule six user level threads (small circles) on three virtual processors (large circles) running on three physical processors. The thread library creates kernel threads for the three virtual processors and parks two additional spare threads in kernel. This is implemented by creating kernel threads which execute a special system call to block in kernel on an application specific queue. Three user level threads are executing on the virtual processors (Figure 3.2). The arrows indicate the binding of a user level thread to a kernel thread or the binding of a kernel thread to it's state in kernel mode (shown as squares). A kernel thread, which isn't bound to one of this states, is running either in user or kernel mode. The numbers identify the physical processor a kernel thread is running on.

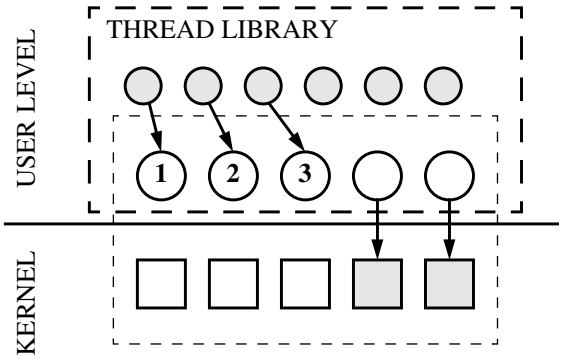



**Fig. 3.2: Parked Sleeping Threads**

When a user level thread blocks in kernel, the state of the corresponding kernel thread changes to *blocked Sleeping Thread* (**SB**). One of the spare threads[2] is deblocked and returns from it's system call. The returned spare thread calls the user level scheduler and chooses a runnable user level thread for execution. This mechanism is very efficient because only a normal context switch (which is necessary in any case when a thread blocks) is necessary. In this situation we can shortcut the normal scheduling algorithm of the kernel and switch directly to the spare thread. From the application point of view nothing has changed. The application still runs on three virtual processors. To reach the situation shown in figure 3.3, the thread library has to be informed about the new state - *blocked in kernel* (**BK**) - of the user level thread. The most efficient way is the use of shared memory pages between kernel and user level library. The kernel itself can manipulate the state of a user level thread. The kernel routines dealing with this data structures have to be robust enough to ensure the integrity of the kernel even in the case of wrong or inconsistent data in the shared pages.

2. The case of no available spare threads is discussed later.

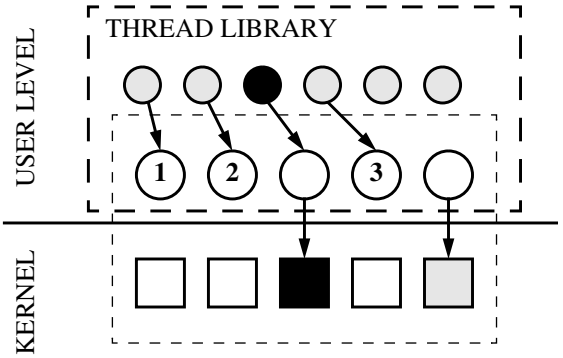



**Fig. 3.3: Blocked kernel thread replaced by a spare thread**

When a thread is deblocked again, it should not be scheduled instantly. This would result in four running kernel threads for the application and therefore in a kernel scheduling of the virtual processors. The kernel thread changes to the state *deblocked Sleeping Thread* (**SD**) and again the thread library has to be informed about a change in state. The user level state changes to *deblocked in kernel* (**DK**) by manipulation of the shared data (Figure 3.4). The thread library can now decide to reschedule the deblocked thread.

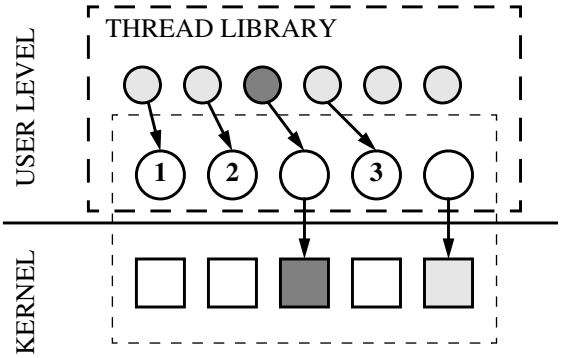



**Fig. 3.4: Deblocked kernel thread**

In figure 3.5 the virtual processor running on physical processor 2 chooses the deblocked thread for execution. A normal user level context switch can't be performed, be-

cause the thread is still in kernel mode. After saving the context of the running user level thread the virtual processor executes a system call to block in kernel as a spare thread. The identification of the thread to be scheduled instead is passed as a parameter to the kernel. The kernel switches, again shortcutting the normal algorithm, from the calling thread to the deblocked virtual processor. The system is in the initial state again. Three virtual processors are running and two spare threads are parked in kernel.

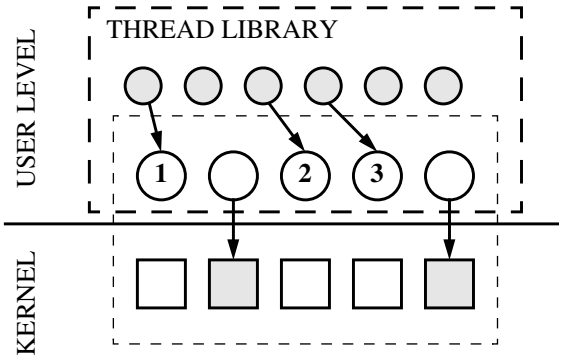



**Fig. 3.5: Deblocked thread continued from user level**

In this scenario the physical processor executing the third user level thread has changed after the deblocking. The thread started on processor 3 and now runs on processor 2. But it is straight forward to implement a scheduling strategy with processor affinity. The strategy has to keep track of the spare threads replacing blocked virtual processors. This works without detection of the actual processor id, but of course only in absence of kernel scheduling. In the other case we need a call to get the processor id in order to implement affinity.

### 3.3 Strategies and exception handling

We distinguish between two strategies: The scheduling strategy and the *Sleeping Threads* strategy, responsible for managing spare threads. Only the second is of interest for the scope of this paper. It has to ensure that at least one spare thread is available at any time. Like a scheduling strategy the *Sleeping Threads* strategy is called by the scheduled threads themselves at time of context switches. But with the creation of additional spare threads we run into problems. The kernel must not schedule the new thread until it is needed as a spare thread to prevent kernel scheduling. To solve this we need a special system call for thread creation which enqueues the new thread as a spare thread.

If a thread blocks in kernel and no spare thread is available, we need a method to run the strategy at user level to install new spare threads. Because we can´t guarantee an interruptible virtual processor to run the strategy, we need an additional thread blocked in a special system call. This thread is deblocked and scheduled on the processor of the blocked kernel thread. It runs the strategy to install at least one spare thread, executes the system call again, blocks and passes it's processor to a new spare thread. All the exception handling is done on the same processor. No kernel scheduling between other virtual processors is necessary but this process is much slower than simply launching a existing spare thread.

### 3.4 Implementation

A first implementation is in work for the Convex Exemplar Architecture. The Convex Exemplar is a NUMA architecture with up to 128 HP RISC 7100 processors. The operating system is based on OSF/1 AD [7]. As a MACH based system the Convex is very suitable for an implementation of the *Sleeping Threads*. Shared pages between kernel and user level and shortcutting the normal scheduling algorithm (using the MACH mechanism for handoff scheduling) is easy to implement. We plan to use task (or process) specific queues for parked and deblocked *Sleeping Threads*. Kernel threads, which are in one of these states, are dequeued from the normal queues of the scheduling system. Only blocked *Sleeping Threads* have to be handled in the same way as normal threads in order to be found by the wakeup algorithm. To activate the new scheduling mechanism for a thread the *Sleeping Thread* flag has to be set in the thread status. All other threads will be scheduled by the standard system algorithm to ensure compatibility of the modified operating system with old applications.

### 3.5 Sleeping Threads and MACH

As stated above MACH is very suitable for implementing a mechanism like the *Sleeping Threads*. But MACH also conflicts with the concepts behind the *Sleeping Threads*. The UNIX emulation of MACH uses the MACH message system for nearly every system call. The application thread sends a message to the UNIX server and blocks in the MACH kernel waiting for the result. A server thread executes the system call. With the *Sleeping Threads* we release a spare thread to substitute the blocked virtual processor and run with a constant amount of threads. This results in an additional running thread inside the server. But this is a common problem of the MACH system. We can even get in conflict with server threads running for other applications. On a NUMA system like the Convex the interfering application can run on a different node and therefor a different set of processors. The only way to solve this is to reserve different sets of processors for applications and servers. For applications with only few calls to the system this would result in idle processors. To address this we need a dynamic processor allocation integrated in the *Sleeping Threads*.

The *Sleeping Thread* mechanism allows to access one of the main problems of MACH based systems. Every system call handled in the server results in the blocking of the calling thread even if the server thread can handle the call without blocking (e.g. non blocking system calls or buffer cache hits). After completion of the server call the calling thread is enqueued as runnable and has to wait to be scheduled again. Because of shifting all scheduling decisions to the user level thread library, we can handle such threads with a

higher priority. The freed spare thread could first check whether the blocked thread is runnable again and give control back to the deblocked virtual processor instead of choosing a different user level thread to run. After the completion of the server call the calling thread maybe continues on the same processor with a hot cache.

The Convex Exemplar operating system has a lot of changes and optimizations to address performance problems. Instead of MACH messages faster remote procedure calls into the server are used. When ever possible the rpc is executed by a server thread on the processor the calling thread was running on giving control back in the same way. The *Sleeping Threads* have to be adapted to a more sophisticated MACH like the Convex OS.

## 4 Related work

The conflict between an efficient thread switching on the one hand and an insufficient system integration on the other hand can be found in nearly all available thread libraries. We want to mention the Scheduler Activations, which are ported to MACH [8], and the First Class User Level Threads as further attempts to address this problem.

Our attempt is similar to the Scheduler Activations. The main difference is that the kernel creates new Scheduler Activations which execute upcalls to the user level. This can´t be as efficient as simply freeing a blocked existing thread. The Scheduler Activations, as stated by Anderson et. al., are not designed for NUMA architectures. One problem is an inherent kernel scheduling in case of deblocking a thread. A processor is needed to execute an upcall to inform the thread library about this event. Therefore another Scheduler Activation has to be interrupted by the kernel, even if it currently runs a higher prioritized thread.

The First Class User Level Threads address the special requirements of NUMA architectures by guaranteeing the execution of a virtual processor always on the same physical one. Kernel and user level is integrated in a high degree to a common scheduling. This allows to implement most of the kernel thread features for user level threads. But this also needs a lot of work in kernel which is related to the used operating system (PSYCHE).

The aims of the *Sleeping Threads* have some different aspects: Doing as few as necessary in kernel to build a mechanism to allow further work and research in NUMA related scheduling and thread libraries. This is achieved by an implementation which shifts all scheduling from kernel to user level.

## 5 Summary and future work

The *Sleeping Threads* are a mechanism which allows to solve some problems found in existing user level thread libraries. By parking spare threads the thread library is able to react on threads blocked in kernel. The number of virtual processors remains fixed. All scheduling inside an application is shifted into the user level, even in case of a blocked thread in kernel. This allows a user level scheduling with respect to the architectural features of modern parallel computers (e.g. memory and cache architecture) without interfering context switches from kernel. The good system integration is achieved by an extended communication between thread library and kernel. We think that this attempt is suitable to implement further kernel thread features at user level (e.g. signaling specified threads). An important advantage of the extended communication is that kernel related system extensions could be implemented mainly at user level with minor kernel changes.

The aim of the future work in this project is to look at the effects of a communication between user level and kernel in the area of parallel programming on shared memory architectures (especially NUMA architectures). In which way will the described mechanism, an extended communication or the transfer of more multiprocessing support from kernel to the user level effect efficiency and the capabilities of the programming model “threads”? The next steps are an implementation of the *Sleeping Threads* and the adaptation of thread libraries to the new mechanism. Based on this implementation we want to look at the characteristics of *Sleeping Threads* applications. One point of interest will be the comparison of different scheduling strategies with respect to the cache use on NUMA architectures.

## 6 References


[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. M. Levy, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, ACM Transactions on Computer Systems, vol. 10, no.1, 1992, pp. 53-79

[2] T. J. LeBlanc, E. P. Markatos, B. D. Marsh, M. L. Scott, “First-Class User-Level Threads”, Operating Systems Review, vol. 25, no. 5, 1991, pp. 110-121

[3] “SunOS 5.3 Guide to Multithread Programming”, Sun Microsystems, Mountain View, 1993

[4] J. K. Ousterhaug, “Scheduling Techniques for Concurrent Systems”, Proceedings of the Third International Conference on Distributed Computing Systems, 1982, pp. 22-30

[5] D. Feitelson, L. Rudolph, “Distributed Hierarchical Control of Parallel Processing”, IEEE Computer, vol. 23, no. 5, 1990, pp.65-77

[6] D. J. Black, “Scheduling and Resource Management Techniques for Multiprocessors”, Carnegie Mellon University, Technical Report CMU-CS-90-152, 1990

[7] “Convex Exemplar Architecture, Second Edition”, Convex Press, Richardson, 1994

[8] P. Barton-Davis, D. McMamee, R. Vaswani, E. D. Lazowska, “Adding Scheduler Activations to Mach 3.0”, USENIX Mach III Symposium Proceedings, 1993, pp. 119-136