Porting Multithreading Libraries to an Exokernel System

Ernest Artiaga, Albert Serra, Marisa Gil
Department of Computer Architecture
Universitat Politècnica de Catalunya
c/ Jordi Girona 3, Mòdul D6, 08071 Barcelona, Spain
e-mail: {ernest, alberts, marisa}@ac.upc.es

ABSTRACT

In this paper, we present an extension to allow multithreaded applications to run on an Intel-based exokernel system. For this purpose, we have ported a user-level threads package to such environment. Our goal is to evaluate the suitability of running multithreaded parallel applications on top of a multiprocessor system managed by an exokernel. We present the design of a prototype library, the lessons learned during its implementation and our proposals for this kind of systems.

KEYWORDS: exokernel, Xok, ExOS, CThreads, multithreading, multiprocessor.

Porting Multithreading Libraries to an Exokernel System

Ernest Artiaga, Albert Serra, Marisa Gil
Department of Computer Architecture
Universitat Politècnica de Catalunya
c/ Jordi Girona 3, Mòdul D6, 08071 Barcelona, Spain
e-mail: {ernest, alberts, marisa}@ac.upc.es

ABSTRACT

In this paper, we present an extension to allow multi-threaded applications to run on an Intel-based exokernel system. For this purpose, we have ported a user-level threads package to such environment. Our goal is to evaluate the suitability of running multi-threaded parallel applications on top of a multiprocessor system managed by an exokernel. We present the design of a prototype library, the lessons learned during its implementation and our proposals for this kind of systems.

KEYWORDS: exokernel, Xok, ExOS, CThreads, multi-threading, multiprocessor.

1. INTRODUCTION

Current trends of the operating systems design are oriented to the maximum simplicity in the kernel code. The kernel is the part of the operating system that is mandatory and common to all other software [8]. In the exokernel approach [5], developed at MIT, the kernel simply exports the hardware resources in a secure way, through a low level interface. The high level abstractions and functionalities are provided by library operating systems implemented at user-level. This architecture allows application-specific customization of the operating system services by extending, specializing or replacing libraries, resulting in performance benefits [7].

Parallel applications are specially sensitive to the amount of resources available. The exokernel approach allows applications to have an efficient access to actual hardware resources and to control them. Such access can be exploited by running shared-memory parallel applications directly on top of a multiprocessor exokernel and letting them to adapt to the actual resources, increasing the overall performance.

In this paper, we study the requirements of an extension library to provide user-level multithreading functionalities on an Intel-based exokernel environment. As a proof of concept, we have ported a user-level threads library to Xok, as a first step to use the exokernel technology to run parallel applications on Intel-based multiprocessor machines.

The main goals of our work are:

- To study the suitability of a user-level library to execute parallel multithreaded applications on top of an exokernel system in a comfortable and efficient way;
- And, from the lessons learned, to design and propose a mechanism to provide efficient support to run parallel application on a multiprocessor exokernel-based system.

This paper discusses the design of the multithreading package, its prototype implementation and its interaction with the original exokernel system. We also present the lessons learned from this work and our proposal for a parallel extension of the exokernel. Section 2 presents an overview of the exokernel application architecture, focusing on scheduling issues; Section 3 discusses the design of the CThreads port to the exokernel; Section 4 presents de implementation details; Section 5 describes the tests done to prove the factibility of the project; Section 6 presents the lessons learned and new proposals; and finally, Section 7 describes the future work.

This work has been supported by the Ministry of Education of Spain (CICYT) under contract TIC 98-0511

2. OVERVIEW OF THE EXECUTION AND PROCESSOR MANAGEMENT IN THE EXOKERNEL

The exokernel technology heavily depends on the underlying hardware. The library described in this paper is designed to work on an Intel-based system, and it will use an exokernel environment specifically designed for this platform (called ExoPC and developed at MIT).

In the ExoPC environment, applications are linked to a library (ExOS) which provide the operating system functionalities through conventional procedure calls and, when exokernel services are required, the library performs the necessary kernel calls to the exokernel (Xok). Only Xok runs at *kernel level*: so, the *system level* which provides Unix services, composed of Xok and ExOS, is divided between the *kernel* and *user* levels. Figure 1 shows the ExoPC architecture.

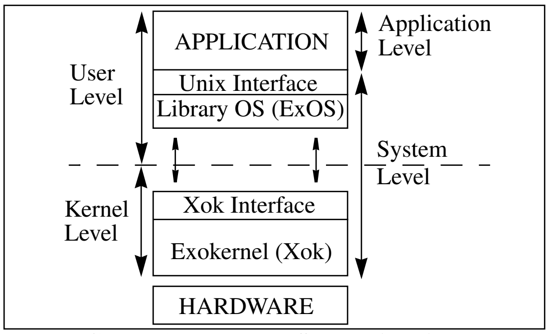

FIGURE 1. Exokernel System Architecture

The only abstraction provided by the exokernel is the *environment*, an entity for secure physical resource allocation. The *environment* keeps a capability list, a series of entry points to notify events to user-level, some resource accounting information and a page table. Part of the *environment* data structure is a user-writable area that allows information exchange between the kernel and the library OS.

The physical processor is exported in terms of a fixed number of time slices or quanta, organized into a circular array. While such circular array is traversed, execution control is consecutively transferred to the *environment* owning the current quantum slot. Non allocated slots are skipped. An *environment* can allocate either random quanta, or quanta located in specific slots which provide the most adequate processor time 'distribution' for the application. This is an interesting mechanism to develop different scheduling policies as real time, time sharing, etc.

The kernel notifies the *environment* about the beginning and the ending of a quantum. Entry points for both upcalls are specified at user-level, and they can change to fit the application needs. The *prologue* and *epilogue* code, called at the beginning and end of the quantum, are responsible for context saving and restoring, and they are executed at user-level.

In a multiprocessor system, each processor has its own local circular array of quanta. Then, environments can allocate quantum slots in specific processors. Nevertheless, Xok prevents an *environment* from 'running' in slots of different processors exactly at the same time. Thus, it does not offer actual parallelism inside a single environment.

In the ExoPC distribution, the library OS (ExOS) uses an *environment* to represent a monothreaded Unix process and allocates a single quantum for it, so that it has a slice of processor time to run. This is enough to provide classical Unix services. Different Unix processes are basically scheduled upon a clock selection mechanism, following the circular array of quantum slots provided by the exokernel.

3. CTHREADS ON THE EXOKERNEL ENVIRONMENT: XOK-CTHREADS

As said before, it its interesting to be able to run parallel applications on top of a Xok-based system, since parallel applications can benefit from the direct access to physical resources in a multiprocessor machine. A common view of a parallel applications consists of a set of cooperating *threads*, sharing a single address space within a *process*.

It is also desirable that a parallel multithreaded application was able to use Unix services as any other application. So, a user-level multithreading package should manage the concurrency inside the application, scheduling the different control flows, while co-existing with the other libraries that may be needed by the application.

To run this kind of applications over the ExoPC, we must face some important aspects:

- *Environments* are currently designed as uniprocessor entities provided by Xok.
- ExOS currently provides support for Unix-like monothreaded *processes* only.

In this paper, we will use the CThreads package [3] to provide user-level multithreading functionalities, while trying to coexist with the Unix Services provided by the ExoPC Environment. This library is widely used in the OSF/Mach microkernel-based systems [1] and we have experience porting it to different parallel environments [6]. From now on, we will call Xok-CThreads the CThreads port to the Xok-based system.

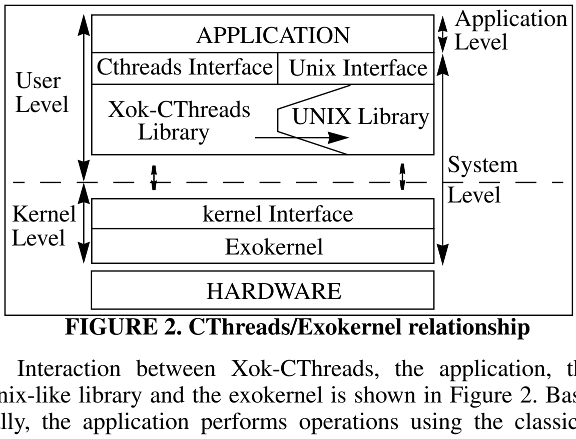

FIGURE 2. CThreads/Exokernel relationship

Interaction between Xok-CThreads, the application, the Unix-like library and the exokernel is shown in Figure 2. Basically, the application performs operations using the classical Unix interface, and it manages multithreading using the CThreads interface. Xok-CThreads intercepts some of the C and Unix calls in order to manage concurrent execution of flows and reentrance. The upcalls from the kernel to the user-level are also captured by Xok-CThreads, which sends the necessary information to the Unix library operating system (ExOS).

In next subsections we explain the basic design principles of the original CThreads package and then we discuss how to port such concepts to the exokernel environment.

3.1. Overview of the CThreads Library

The CThreads package was designed to handle concurrency and parallelism at user-level [3]. It provides primitives for user-level thread creation, synchronization and concurrency management.

A *cthread* object represents an execution flow and it needs a virtual processor to run (originally, CThreads was designed for Mach, using kernel threads as virtual processors). Different *cthreads* can be multiplexed on top of a smaller number of virtual processors. Context switching among *cthreads* is performed by the CThreads library and it is non-preemptive.

A *cthread* can also be *wired* to a virtual processor, *reserving* it. This guarantees that such *cthread* has an available virtual processor to run at any moment (though, on several systems, that does not mean having an actual physical processor to run).

3.2. Porting CThreads to the Exokernel

One of the concerns when porting CThreads to Xok was the notion of virtual processor, which will be located at user-level.

Our proposal consists of using a user-level abstraction (an *execution context* [6]) to implement the virtual processors in Xok-CThreads. An *execution context* extends the per-quantum information and keeps the low level context of a *cthread* (just a few registers and a pointer to the *cthread* itself) when it cannot continue to run on a physical processor (e.g. when the current quantum expires). Each physical quantum allocated by the application has its corresponding user-level *execution context*. An *execution context* is said to be *active* when a *cthread* scheduled on top of it is actually running on a physical processor. Otherwise, it is said to be *deactivated*.

The *execution context* is maintained by the library. It is lighter than a classical virtual processor provided by the kernel. Moreover, it is fully configurable by the application, since it is fully implemented at user-level.

Xok-CThreads uses a two-level scheduling mechanism. The high-level scheduling is non-preemptive and allows context switches among *cthreads* on top of *execution contexts*. Such context switches occur at safe and well known points, and they are related to specific interface routines, such as synchronization primitives, etc.

On the other hand, the low-level scheduling is related to the mapping of the *execution contexts* to the physical processors via the Xok quantum mechanism. Such a mechanism is preemptive, and it can cause context switches at unexpected points. The library is responsible to reach a safe state before switching contexts by using the *epilogue* and *prologue* kernel upcalls.

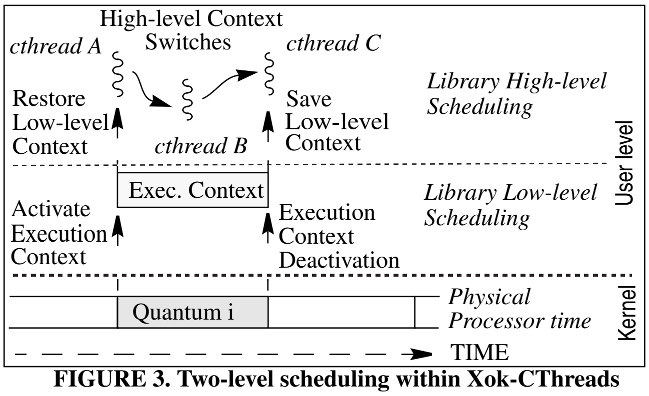

FIGURE 3. Two-level scheduling within Xok-CThreads

Figure 3 represents the different scheduling levels within the Xok-CThreads environment: an *execution context* is activated when its quantum begins, and it is deactivated when such a quantum expires. On top of the *execution context*, different *cthreads* execute concurrently.

It is possible for an *execution context* to become idle. This occurs, for example, when the current *cthread* blocks and there is no other ready *cthread* to make a high-level context switch. In such situation, the low-level scheduling mechanism tries to choose another *execution context* (associated to another quantum) to restore it to use the remaining time in the physical quantum. So, the mapping between an *execution context* and its corresponding physical quantum is not completely strict.

Figure 4 shows the architecture of a Xok-CThreads application with the basic data structures used by the library. Each *execution context* structure represents the *execution context* to be activated for the corresponding physical quantum.

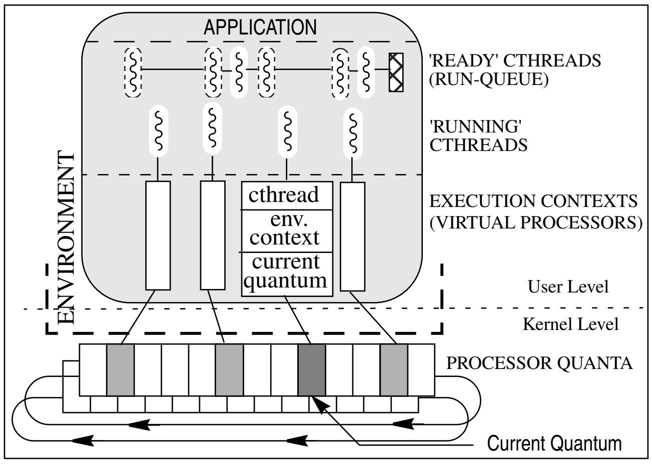

FIGURE 4. Internal structure of a Xok-CThreads application

One of the main differences between virtual processors in other operating systems and Xok-CThreads *execution contexts* is that allocating a quantum in the exokernel guarantees a fixed amount of CPU time to execute every quantum round in the kernel. Instead, having a virtual processor allocated in other systems does not usually provide any guarantees of execution. We consider this an advantage, because applications have a greater knowledge and control of the available physical resources.

Moreover, management of the *execution contexts* is completely done at user-level. This makes it much easier to modify their behavior and adapt them to specific applications and programming models.

4. THE IMPLEMENTATION

In this section we will comment details of the implementation of the Xok-CThreads. The current implementation takes into account the following objectives:

- An application will be able to use the Xok-CThreads interface to have different execution flows.
- The Xok-CThreads library should coexist with the Unix-like library. This means that the code in a *cthread* must be able to use ExOS services.

The following subsections focus on several aspects of the implementation, such as data structures, communication between ExOS and Xok-CThreads, low level scheduling and kernel changes that would improve the performance of the library.

4.1. Preliminary: Semantics of the CThreads interface

In Xok-CThreads, a virtual processor is instantiated by an *execution context*. Currently, an *execution context* is a user-level structure closely related to a physical processor quantum as exported by the kernel, so that it actually represents a portion of the physical processor time.

Every time a new *cthread* is created, the library allocates a new quantum and creates a new *execution context*, which is set up to run the new *cthread* on it. The library internally decides the physical processor and the specific quantum slot to allocate (though this behavior may be overridden by the application).

The application may also limit the number of *execution contexts*. Then, some *cthreads* can be ready to execute but not assigned to any specific *execution context*. They will be put into execution through a high level context switch when a running *cthread* cannot continue (e.g. for synchronization operation).

The application can also wire a *cthread* to its current *execution context*, and to the current quantum slot. This prevents other ready *cthreads* from using that *execution context* when the owner *cthread* blocks. Moreover, if the *execution context* is deactivated before finishing its quantum, the remaining time cannot be used by a different *execution context*; instead, it will be accumulated for the next round.

The wiring mechanism guarantees that the wired *cthread* owns a portion of the physical processor time (somehow, the *cthread* has reserved a portion of the physical processor), so that it will be able to run again as soon as it is unblocked, despite the number of *cthreads* fighting for an available *execution context* (and the corresponding quantum) to run on. This mechanism is specially useful when there is a need to ensure that a *cthread* will have processor time available on a regular basis. It can be used by applications with specific-work threads, as multimedia applications that have to provide a certain quality of service, scheduler threads, etc.

4.2. Data Structures

Xok-CThreads uses an *execution context* for each physical quantum allocated by the application. The information needed to manage the *execution context* is kept in a structure that contains, basically, the physical cpu and quantum identifiers, room for the FPU context, and pointers to the high level structures: the current cthread structure and the current stack pointer for that *cthread*. The rest of the low level context (basically the register set) is saved in the *cthread* stack.

The library also maintains a map for each physical quantum allocated by the application with a pointer to the corresponding *execution context*. When an upcall indicates the beginning of a new quantum, the service code checks that table to know which *execution context* is to be restored. The low level data structures are shown in Figure 5.

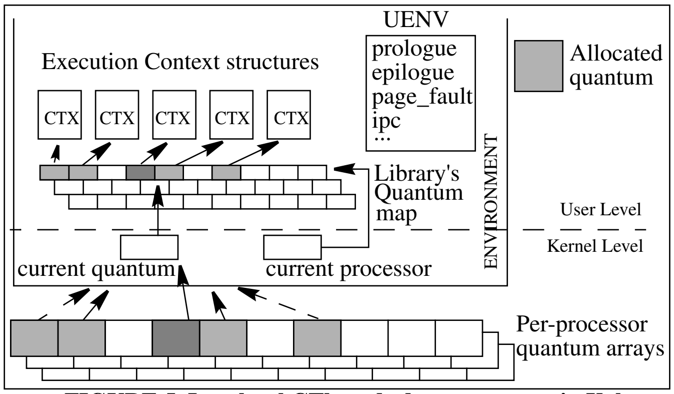

FIGURE 5. Low level CThreads data structures in Xok

4.3. Low-Level Scheduling

High-level scheduling in the CThreads library is very simple. There is a global queue of ready *cthreads*. Basically, when a *cthread* blocks or finishes, the next *cthread* in that ready queue is taken as the next *cthread* to run on its *execution context*.

The low-level scheduling, instead, is responsible for managing not the *cthreads*, but the *execution contexts*. It uses two basic structures: the quantum map and the context queue. Both the structures are replicated for each physical processor.

When a new quantum begins in a processor, the prologue code is invoked. The prologue checks for the current processor and quantum, and it uses this information to find the associated *execution context* in the quantum map structure (Figure 5). Then, the corresponding low-level context is restored and the high-level *cthread* resumes its execution. From this point, the following things may occur to the *execution context*:

- The quantum expires and the *execution context* is to be preempted.
- The *cthread* running on top of the *execution context* wants to yield the processor.

A quantum expiration means that the running *cthread* is involuntarily preempted, which is the same to say that such *cthread* is ready to go on executing as soon as possible. In this case, the epilogue routine saves the low-level context of the *cthread* within the corresponding *execution context* structure and deactivates it.

The aforementioned *cthread* can be resumed during a different physical quantum. To allow for this, the *execution context* is inserted at the head of a per-processor context queue, in order to get restored as soon as any quantum is available for the application (and its corresponding *execution context* has no work to do). The goal of inserting at the queue's head is restoring the last preempted *cthread*, so that its data have a chance to be still in the cache. Nevertheless, if the *cthread* was wired, it will have to wait till the very same quantum slot during the next quantum round.

A different situation occurs when the *cthread* on top of an *execution context* wants to yield the processor. At high-level, this can be due to two possibilities: the *cthread* has to block and there is no other *cthread* in the ready queue or, simply, the *cthread* voluntarily wants to yield the processor because it has no better thing to do. In any case, it is not a good idea to try to go on executing the *cthread*, so the corresponding low-level *cthread* state is saved and the corresponding *execution context* is deactivated.

Then, the remaining physical quantum time can be used to activate a different *execution context*. So, the per-processor context queue is checked to look for ready *execution contexts* (if there is no *execution context*, a system trap will eventually yield the processor). Finally, the yielding *execution context* is also queued int the per-processor context queue. As we are not specially interested in running this cthread as soon as possible (in fact, the *cthread* did not want to run), insertion is done at the tail of the queue. This handoff mechanism is specially useful when dealing with synchronization (barriers, etc.). The waiting *cthreads* may block and let the *execution context* of the worker to use its quantum, so that it finishes earlier and the waiting cthreads may resume execution.

An extension of this low-level scheduling mechanism consists of allowing work-stealing among different physical processors. A processor can look in other processors queues for ready execution contexts when its own ready queue is empty. This feature has not yet been implemented in the Xok-CThreads prototype.

4.4. The Neighborhood: Handling with ExOS

Xok-CThreads should be able to coexist with ExOS (the library OS which provides Unix services in the ExoPC). For that, Xok-CThreads should be ExOS compatible when saving and restoring low-level contexts and managing the FPU. Also, Xok-CThreads must share information about low-level events (quantum expiration, etc.) with ExOS. Multiprocessor management is also affected by the relationship between ExOS and Xok-CThreads.

Xok allows the user-level (an application, or a library) to specify user-level routines to be invoked when a scheduling event occurs. ExOS uses such entry points to provide the Unix flavor (signal delivering, paging, ipc, timeouts, etc.). Xok-CThreads need to override them for low-level scheduling (*execution contexts* management) but, nevertheless, Unix services provided by ExOS should also be active.

In order to ensure the coexistence between both systems (using *cthreads* and unix processes simultaneously) Xok-CThreads entry point routines pass the necessary information to the ExOS library. This is done essentially at three points: prologue, epilogue and yield routines. Currently, Xok-CThreads implementations of such routines explicitly call the corresponding ExOS routines to let it know about the scheduling events.

ExOS also assumes a process-based programming model. It uses a Xok *environment* to implement a monothreaded Unix process. Such process uses a single user stack, which is used to save and restore the process context (cpu registers, etc.).

Xok-Cthreads requires a separate stack for each *cthread*, and the *cthread* to be executed when a quantum begins depends on the active *execution context*. So, when a context is restored, a scheduler code from Xok-CThreads is executed to choose the next *cthread* to run and switch to its stack to restore the corresponding *process-like* context. This stack switching is transparent to ExOS.

Xok-CThreads also needs to maintain more than a single FPU state, since several *cthreads* may have been interrupted with valid FPU contexts. So, the library keeps that information within the *execution context* data structures.

Finally, considering the multiprocessor management, it is important to note that ExOS internally assumes that a Unix process uses a single Xok *environment*. This is important because Xok provides the *environment* as a *sequential* abstraction: it cannot be active on more than a single processor simultaneously. With the current implementation, a parallel application must be supported by several Xok environments. However, ExOS does not handle such application as a Unix process. This means that an application using Xok-CThreads and ExOS simultaneously cannot be *parallel*, but simply *concurrent*.

5. TESTS

We have tested our prototype implementation of the Xok-CThreads library on an Intel Pentium-based machine. Tests consisted of synthetic applications to observe the behavior of basic features, and some applications and kernels from the SPLASH-2 suite to see the library behavior for 'normal' applications. The goal of these tests was not to measure performance, but to check the correctness of the implementation and to look for bottlenecks and performance cliffs due to library overhead. The applications used both ExOS and Xok-CThreads services and, therefore, they were executed on a uniprocessor environment.

The following table shows the execution times for some SPLASH-2 applications: Barnes, Raytrace and LU. These applications were chosen because they fit in memory and they last for a significative amount of time in our machine. There are two versions of each application: one of them uses several ExOS processes, while the other one uses a single ExOS process with several *cthreads* in a single address space. Both versions were executed using 1, 2 and 4 execution flows (implemented with Unix processes or *cthreads*, depending on the version). The execution times are in seconds.

	1 flow		2 flows		4 flows	
	ExOS	CThr.	ExOS	CThr.	ExOS	CThr.
Barnes	37.92	36.50	39.62	36.46	43.04	36.79
Raytrace	235.64	242.34	240.03	254.19	251.51	260.12
LU	53.37	54.49	53.42	54.52	53.88	54.56

TABLE 1.

The execution times from executing the applications with our Xok-CThreads implementation are comparable to the results obtained using multiprocess versions of the applications using the ExOS services only. This means that our library does not introduce a great loss of performance, while it provides a comfortable multithreaded programming model to the application programmers.

6. LESSONS LEARNED AND PROPOSALS

In this section, we summarize some facts about the ExoPC environment which have had consequences on our CThreads implementation for Xok. Eventually, we make some proposals for future versions which could improve the flexibility of the system and its adaptability to multiprocessor systems.

6.1. Lessons Learned

The exokernel is a very flexible system. This fact allows us to build powerful application-specific libraries to improve the behavior of certain programs, without disturbing the rest of applications. For example: in this work, we provide a multithreaded environment just for applications which require it. Xok allows this kind of extensions by exporting hardware and kernel information to user-level, so that the application (or the library) has enough information to take 'intelligent' scheduling decisions based on resource availability and system status.

User-level libraries providing system services should be designed keeping in mind that they will probably have to coexist with other libraries. So they should provide a flexible interface to exchange data with other libraries. For example, both Xok-CThreads and ExOS libraries must be aware of some kernel events (such as quantum beginning and exhaustion); so, Xok-CThreads captures such events, but it also must ensure that the resulting state (contents of stack, etc.) is consistent with what ExOS expects to find.

Since the exokernel exports the hardware, the libraries must be aware of the details of the architecture. For example, the context switch code in Xok-CThreads must deal with the FPU unit in an Intel Pentium processor. The code which handles with the FPU must be not only correct, but also efficient. It is important to know when such context actually needs to be saved or restored, in order to avoid unnecessary floating point exceptions, which are usually time expensive.

6.2. Proposals

The Xok exokernel has proved to be flexible and robust, and it provided enough support to implement a multithreaded library easily. Nevertheless, we propose some changes to improve that support, specially concerning the multiprocessor management and multi-library coordination.

6.2.1. Multiprocessor support

The current multiprocessor version of Xok has evolved from an uniprocessor version without major design changes. Basically, processor management structures have been replicated at kernel-level. Nevertheless, from the user-level point of view, few things have changed. The *environment* abstraction enforces an uniprocessor vision of the system: only information about the current processor is available to user-level, and the *environment* can be active in a single processor at a time. Also, IPC mechanisms are not thought to communicate environments simultaneously “running” on different processors.

In this scenario, a shared-memory programming model can still be implemented by mapping a parallel application onto several Xok *environments* sharing the address space (each *environment* representing an execution flow). However, it would be more comfortable having an *environment* which give access to the multiprocessor resources of the hardware. For this, it would be interesting having support for the following features:

- Current *environments* are implicitly associated to an execution flow. Instead of that, it is more convenient to see an *environment* as a resource container, so it can have quanta allocated in different processors. Several execution flows (threads, processes, or any other thing) can use the resources of the very same container and run in parallel on different processors.

  In the same way, the current semantics of 'blocking' an *environment* till certain event occurs should be changed by 'blocking' an execution flow. For example, an execution flow may block for an I/O operation but the rest of flows in the environment should be able to go on working.
- So, *Environments* should be able to be active on several processors at the same time. The *environment* entry points related to scheduling (prologue, epilogue) could be specified in a per-processor basis (or even in a per-quantum basis), so that different code can be executed in different processors in a natural way.
- Kernel information about the use of the processors must be visible from user-level. Such information can be needed by a scheduling user-level library to decide in which processor to run a particular execution flow. Such information would include the current quantum in each processors, the remaining ticks on each quantum, etc.
- The current exokernel IPC is synchronous and performs a processor handoff to the target *environment*. This mechanism assumes that the target will be ready to run in the same processor. This is not necessarily true in a multiprocessor system, so IPC should be designed to work efficiently when the target *environment* is already running in a different processor (or even in multiple processors, if we consider the possibility of multiple execution flows). Supporting scheduler activations [2][6][9] and multithreading within the *environment* is probably an interesting approach for IPC.

Additionally, libraries providing services on top of a multiprocessor exokernel system should be reentrant, since a multiprocessor environment could be involved in more than one library call simultaneously.

6.2.2. Libraries

Libraries should be designed to coexist with other libraries in the same system. Different services may require knowing about kernel events. For example, scheduling-related libraries need to know about quantum-related events, such as beginning and expiration. Instead of overwriting those entry codes, libraries should register their routines to be invoked at such points. This would allow code from different libraries to be executed, for example, when a quantum expires. Then each library can save the required context before returning control to the kernel.

Another interesting point is to design the libraries following not a process-based model, saving the context in the stack, but a continuation-based model. This model reduces the size of the contexts and enforces the independence between different libraries. The required state information would be saved in each library private data structures, instead of the stack. This approach can provide more flexibility when entering and exiting the kernel [4].

Finally, the exokernel interface makes it specially suitable to implement libraries based on the scheduler-activation concept [2]. The execution contexts used for Xok-CThreads are based on this model. *Scheduler activations* have shown to be efficient even in uniprocessor environments [9], and they are easy to use in multiprocessor environments. So, a general-purpose library operating system based on this mechanism would be interesting.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented and discussed some questions about the implementation of a multithreading library for the Xok exokernel. The goals mentioned in the introduction have been fulfilled: we have extended the functionality of the ExoPC system without changing the kernel, we have seen that it is possible to execute thread-based parallel applications on top of the exokernel and we have made some proposals in order to improve the multiprocessor support of Xok.

The ExoPC system, which consists of the Xok Exokernel and the ExOS Library Operating System has shown to be a flexible system. Its simplicity makes it easy to extend the system with per-application customized features, via user-level libraries.

Specifically, our extension allows parallel applications to be multithreaded, while still using the Unix-like services provided by the ExOS library. Our tests show that the execution times when using our multithreading library are very similar to the ones obtained by multiprocess executions based on ExOS. This means that our multithreading library does not introduce remarkable overhead, despite the apparently more complex scheduling mechanism. Moreover, we are providing a tool for the multithread programming model, which is not originally supported by ExOS.

There are still a number of things left to do as a future work. Our CThreads implementation for Xok must be tuned in order to improve its efficiency. Also, more experiments must be done in order to check its behavior under different circumstances. An interesting experiment would consist of measuring the performance of a CThread-based multithreaded parallel application in a non-dedicated medium/heavy loaded environment. The fine grained scheduling of our CThreads implementation should improve the application performance in such circumstances, but this still have to be proved.

We also plan to go on working in Xok modifications to allow for a greater support of multiprocessor mechanisms. A key new feature consists of making the multiprocessor scheduling information available from user-level (simply by mapping it to a user-readable memory area). This will allow scheduling libraries to take more adequate decisions at a lower cost.

Also, the *environment* abstraction should be separated from the notion of execution flow in order to be just a resource container (as the exokernel philosophy states). Such separation will make it easier to use a Xok *environment* to implement objects different than a classical monothreaded Unix Process.

8. ACKNOWLEDGEMENTS

Thanks to Nacho Navarro, Xavier Martorell and the people of the GSOMK team at the Department of Computer Architecture for using their time to comment, discuss and argue about the topics of this paper. They have provided lots of ideas which have been used during the elaboration of this paper.

9. BIBLIOGRAPHY

[1] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M. 1986. “Mach: A new Kernel Foundation for UNIX Development”. Proceedings of the Summer 1986 USENIX Technical Conference, June 1986, pp. 93-112.

[2] Anderson, T.E., Bershad, B.N., Laxowska, E.D. and Levy, H.M. 1991. “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”. In 13th ACM Symposium on Operating Systems Principles, October 1991.

[3] Cooper, E.C., and Draves, R.P. June 1988.“C Threads”.CMU-CS-88-154, School of Computer Science, Carnegie Mellon University.

[4] Draves, R.P., Nershad, B.N, Rashid, R.F., and Dean, R.W. 1991. “Using Continuations to Implement Thread Management and Communication in Operating Systems”. In 13th ACM Symposium on Operating Systems Principles, October 1991.

[5] Engler, D., Kaashoek, M.F., O'Toole, J. 1995. “Exokernel, an operating system architecture for application-level resource management". In XV ACM Symposium on Operating System Principles (SOSP), Copper Mountain Resort, Colorado.

[6] Gil, M., Martorell, X. and Navarro, N., 1995. “The eXc Model: Scheduler-Activations on Mach 3.0”. ISMM International Conference on Parallel and Distributed Computing and Systems, Washington D.C.

[7] Kaashoek, M.F. et Al. 1997. “Application Performance and Flexibility on Exokernel Systems”. In 16th ACM Symposium on Operating Systems Principles, Saint Malo, France.

[8] Liedke, J. 1995. “On Microkernel Construction”. In XV ACM Symposium on Operating System Principles (SOSP), Copper Mountain Resort, Colorado.

[9] Small, C. and Seltzer, M. 1995. “Scheduler Activations on BSD: Sharing Thread Management Between Kernel and Application”. Technical Report TR-31-95, Harvard University.