Running Multithreaded Applications in Exokernel-based Systems: Porting CThreads to Xok

Ernest Artiaga and Marisa Gil
Department of Computer Architecture, Polytechnic University of Catalonia (UPC)
Jordi Girona 3, E-08034 Barcelona, Spain
{ernest, marisa}@ac.upc.es

Abstract

Exokernel-based systems provide efficient access to the system actual hardware resources. Parallel applications can take advantage of such kind of access and adapt to the actual resources available to increase performance. 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 final goal is having a multiprocessor exokernel version to be able to run parallel applications on top of it. We use the exokernel interface to have access to the physical execution resources and we have designed the lower layer of the multithreading library to use them.

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

1. Introduction

Parallel applications are specially sensitive to the amount of resources available. The exokernel approach [5], developed at MIT, allows applications to have an efficient access to actual hardware resources and to control them [7]. 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 describe the port of a user-level threads library to an exokernel, as a first step to run parallel applications on exokernel-based multiprocessor machines. From the experience of implementing a prototype library, we also present some proposals for the design of libraries to provide services for exokernel-based systems.

This paper is organized as follows: 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 feasibility of the project; Section 6. presents the lessons learned and new proposals; and finally, Section 7. describes the conclusions and future work.

2. 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 [7]).

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). The next figure shows the ExoPC architecture.

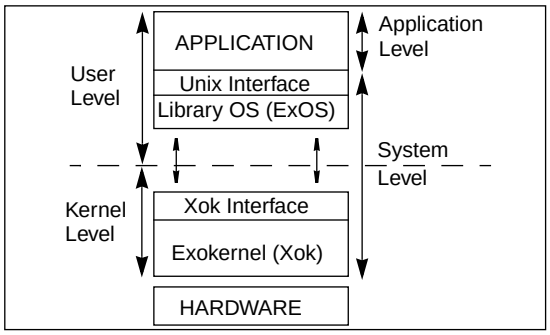

Figure 1. Exokernel System Architecture

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 quanta in specific slots to get the most adequate processor time distribution for the appli-

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

cation. This is an interesting mechanism to develop different scheduling policies as real time, time sharing, etc.

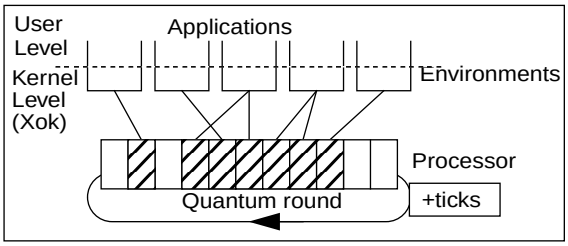

Figure 2. Assigning quanta (processor time) to applications in Xok

The kernel notifies the *environment* about the beginning and the ending of a quantum by executing the *prologue* and *epilogue* code. Such entry points are application-specific. They are also 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 Unix process and allocates a single quantum for its unique thread of control, 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. Running Multithreaded applications in Xok: Xok-CThreads

Nowadays, parallel applications are easy to find in all areas of computing (scientific computation, servers, etc.). A common view of a parallel application consists of a set of cooperating *threads*, sharing a single address space within a *process*. This model results more efficient (and, usually, more comfortable) than using heavy-weight processes to implement parallelism. Moreover, parallel applications can also benefit from the direct access to physical resources provided by an exokernel, specially in multiprocessor systems.

Our goal is to design and implement a library to allow multithreaded applications to run on the exokernel, using both multithreading and classical Unix services. So, we need to extend the single-thread uniprocessor Unix-like environment currently provided by the ExoPC.

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. The choice of CThreads is due to several reasons: its interface is very similar to the standard Posix Threads library, so that it would be easy to extend the port to provide the PThreads interface; moreover, 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.

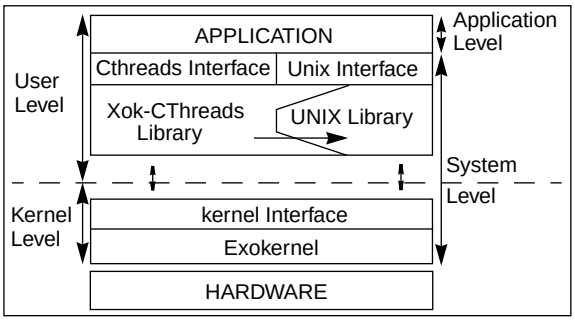

Figure 3. CThreads/Exokernel relationship

Interaction between Xok-CThreads, the application, the Unix-like library and the exokernel is shown in Figure 3. 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 library 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).

It is important to note that only the exokernel (Xok) runs at *kernel level*: so, the *system level* (which provides, for example, the Unix services) is divided between the *kernel* and *user* levels.

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. Principles 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 [1], 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. Virtual processors on the exokernel: the execution contexts

The CThreads interface has been kept when porting the library to the ExoPC architecture. The main concern was the notion of virtual processor. Instead of using a kernel-based abstraction (like a kernel thread), we propose using a user-level abstraction (an *execution context* [6]) to implement virtual processors for 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). Essentially, an *execution context* is the one who knows 'what to do' when a a physical quantum is selected. So, each physical quantum allocated by the application has its corresponding user-level *execution context*.

Nevertheless, the mapping between an *execution context* and its corresponding physical quantum is not completely strict: an *execution context* may become idle (no runnable *cthreads* are scheduled on it) and give its remaining quantum time to a different *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*.

Xok-CThreads schedules *cthreads* on top of *execution contexts* in a non-preemptive way. 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 *execution contexts* are mapped 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.

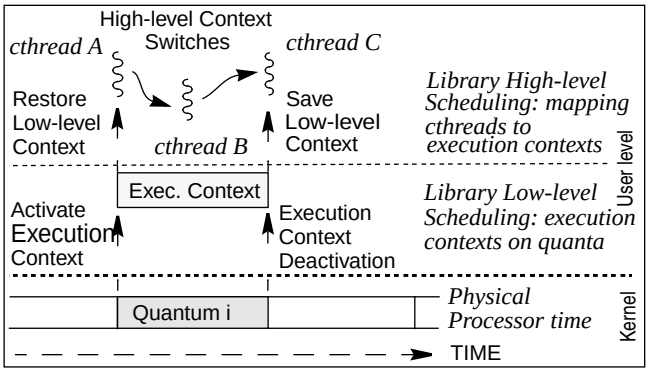

Figure 4. Relationship between cthreads, execution contexts and processor quanta

In summary, an *execution context* is activated when its quantum begins, and it is deactivated when such a quantum expires (or there is no more work to do). On top of the *execution context*, different *cthreads* execute concurrently. Such behavior is represented in Figure 4.

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 means that it is easy for any application to modify them to adapt to its specific needs. By being implemented in a user-level library, such changes will not affect the rest of the applications in the system, and they will not have to deal with kernel complexities which could affect the performance and even the integrity of the entire system.

4. Implementing Xok-CThreads

The Xok-CThreads library has to provide multithreading features to parallel applications. This is an extension to the intel-based exokernel system. Nevertheless, we must take into account that there are other libraries providing other services in the system (for example, ExOS provides classical Unix functionality). So, Xok-CThreads has been implemented keeping in mind that it will have to coexist with other libraries (specially, ExOS).

4.1. Interface and functionalities

Every time a new *cthread* is created, the library allocates a new quantum and creates a new *execution context* (see Section 3.2.), 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.

This behavior may be overridden by the application in order to have in order to use the available processors in a more efficient way. For this reason, we have added the following routines to the Xok-CThreads interface:

- *cthread_current_processor()*: returns the current physical processor identifier.
- *cthread_last_processor()*: returns the last physical processor on which a *cthread* ran.
- *cthread_current_quantum()*: returns the current quantum active in the current physical processor.
- *cthread_fork_on()*: similar to the original *cthread_fork()* (which creates a new *cthread*). This call let the application specify the physical processor and the quantum slot to allocate when a new execution context is created.

An application may also create a *cthread* to run on an existing *execution context*, instead of allocating a new quantum. So, the application can limit the allocated quanta.

On the other hand, an application can 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 implements a processor time reserve mechanism. A wired *cthread* will be able to execute despite the number of *cthreads* fighting for an available *execution context* (and the corresponding quantum) to run on. Such a mechanism is suitable for real-time or multimedia applications, QoS, scheduler threads, etc.

4.2. Library data structures

The CThreads library uses two basic 'high-level' data structures: the cthread_status, which has global data about the library status, and the cthread, which contains management information for a specific *cthread*. During operation,

cthreads are linked together by means of a set of queues and lists, which help to maintain the scheduling policies. Figure 5 shows the basic high level structures in the library.

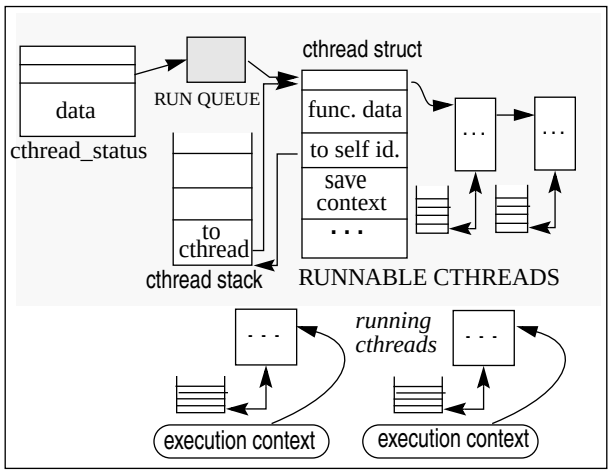

Figure 5. Basic high level CThreads structures

The cthread structure has room to place the high level context and the state of the *cthead* (running, runnable, waiting, detached, ...). Each *cthread* has its own stack, properly sized and aligned, which is also used by "running" *cthreads* (those on top of an *execution context*) for self-identification, since it contains a self-pointer at the bottom. Such *cthreads* are not queued anywhere: they are just pointed from the corresponding *execution context*.

In addition, the Xok-CThreads library keeps the information needed to manage the *execution context* in a new 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 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). All *execution contexts* in a processor are also linked in a queue, to ease its management.

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 current quantum and processor information are available from the user read-only area of the Xok *environment*. The low level data structures are shown in Figure 6.

4.3. Managing execution contexts

Xok-CThreads uses two basic structures (the quantum map and a context queue) to manage the *execution contexts*. Both structures are replicated for each physical processor, and the *environment* entry points use them to know which *execution context* is to be activated or deactivated.

Xok invokes the prologue code each time a new quantum begins for a processor, running it at user-level. Xok-CThreads uses the prologue code to check for the current processor and quantum. Such information 'points' to the *execution context* related to the physical quantum just beginning (via the quantum map structure - Figure 6). Then, the corresponding state is restored, resuming the execution of the high-level *cthread* that was running on that *execution context* when it was deactivated.

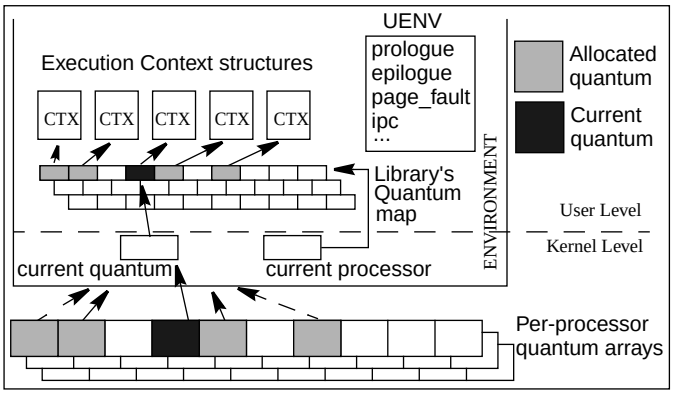

Figure 6. Low level CThreads data structures

An *execution context* will be deactivated when there is no more work to do (i.e. the current cthread has to block or wants to yield the processor, and there are no runnable *cthreads* in the ready queue), or when the physical quantum expires.

In the first case, the running *cthread* is yielding the processor voluntarily. Then, the Xok-CThreads library takes the first *execution context* in the processor's context queue and activates it. The yielding *execution context* is inserted at the end of the queue

In the second case, the running *cthread* is involuntarily preempted due to quantum expiration, which is the same to say that such *cthread* is ready to go on executing as soon as possible. Then, Xok-CThreads uses the user-level epilogue routine to save the context of the *cthread* within the *execution context* structure and deactivates it.

A preempted *execution context* is queued at the head of the processor's context queue. The reason for this is that, in case of a voluntary yield, the last preempted *cthread* will be restored, and its data has a chance to be still in the cache.

The described scheduling mechanism does not apply to *wired cthreads*. A *wired cthread* has a reservation over its *execution context* and its related physical quantum. So, when it yields the processor, the quantum is not used to activate a new *execution context*. Instead of that, Xok accumulates the remaining time for the next quantum round. In the same way, an *execution context* with a *wired cthread* is not eligible to be activated in a different quantum (in fact, it is never inserted in the processor's context queue).

In the general case, the context queue implements a kind of handoff mechanism, which improves the behavior of interthread synchronization (barriers, etc.): a *cthread* may yield the processor, give another *cthread* a chance to do a certain job (even if it is running in a different *execution context*), and resume execution during the same quantum, without kernel intervention.

It is important to note that insertions in the processor context queue do not modify the quantum map structure, which is used by the prologue code to choose the *execution context* to activate at quantum start. The reason is to guarantee that *cthreads* in all the *execution contexts* will have at least a chance to execute during each quantum round,

avoiding the risk of starvation (e.g. in 'busy' waiting code). Then, if the 'official' *execution context* has no work to do, it will use the context queue to activate a new *execution context*. The code to do this operation has no kernel calls, it is short and it has been optimized, so it does not represent a significant overhead.

The current prototype for Xok-CThreads has a context queue per processor, and *execution contexts* do not migrate from one processor to another. Nevertheless, the implementation easily allows a work-stealing scheduling policy among physical processors, so that a processor can look for work in other processors' context queues when its own queue is empty. Anyway, it is clear that the application should be the one to decide which cthreads could migrate and which cthreads could not (inadequate migration may result in having to refill the cache, or causing unnecessary memory access conflicts).

4.4. Memory management

The original CThreads library provided its own code for memory management (specially for stacks and dynamic memory management). Such code was based on Mach 3.0 kernel interface (vm_allocate(), vm_deallocate() and vm_protect()), which provides a number of specific features which CThreads relies on.

In order to maintain the original code unmodified, we decided to build a small library (*libvm*) which implements the vm_* routines by handling the *environment*'s page table through Xok's kernel calls. At this moment, all virtual memory allocated through this library is backed by physical memory and no paging mechanism is provided. This is certainly limited, but enough for test purposes.

The alternative of using ExOS services (the ExOS *brk* implementation) was discarded because it did not provide the required functionality, and for the sake of independence from current Unix implementation (to avoid conflicts with Unix brk, the libvm library uses a different virtual memory range than the one used by the ExOS memory subsystem).

The libvm also serves as an example of how a subsystem can be easily replaced in an exokernel-based system. Parallel applications can take benefit from libraries providing specific functionalities, or replacing general system mechanisms with more simple and efficient code.

4.5. Interaction with Unix services

Multithreaded applications in the ExoPC environment will probably have to coexist with other Unix applications, and even perform Unix system calls themselves. So, Xok-CThreads should be able to coexist with ExOS (the library OS which provides Unix services).

The main interaction points are the *environment* entry points (specially the prologue, the epilogue and the yield code). ExOS uses such routines to provide the Unix flavor (signal delivering, paging, ipc, timeouts, etc.). However, Xok-CThreads need to overwrite such code to manage the *execution contexts*.

ExOS assumes a process-based programming model. It uses a Xok *environment* to implement a monothreaded Unix process. Such process has a single user stack, which is used to save the process context (cpu registers, etc.). Such context is restored during the prologue, after calling the internal ExOS routines to manage paging, signal handling, etc.

On the contrary, 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*.

In order to ensure the coexistence between both systems, the Xok-CThreads entry point routines pass the necessary information to ExOS by calling its system management code directly. Then, the prologue executes a scheduler code from Xok-CThreads 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.

Another element to consider is the FPU state. Since Xok simply exports the hardware, the libraries have to manage it. So, ExOS maintains a per-process (i.e. per *environment*) area to save the FPU state. To reduce the overhead, such context is not restored at prologue, but only when the FPU is used again.

However, Xok-CThreads 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 and transparently updates the ExOS data when necessary.

Note that the FPU status is significant only when the *execution context* is deactivated due to quantum expiration, because its *cthread* could be using the FPU. In other cases, the *cthread* has voluntarily called a routine to yield the processor, so its code won't rely on the previous state of the FPU. This fact is used to restore the floating point status only when required.

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 single Unix process. This means that an application using Xok-CThreads and ExOS simultaneously cannot be *parallel*, but simply *concurrent*.

4.6. Kernel modifications

The implementation of Xok-CThreads did not require kernel modifications. Nevertheless, it would be of great interest having more scheduling information at user-level (which could be located, for example, in the user read-only area of the *environment*).

In order to let an application adapt to the available execution resources, the system should provide, without kernel calls, information about which are the current processor and quantum, and which quanta and processors are available to be allocated by the application.

In summary, physical processor information should be exported by the kernel to make user-level scheduling easier and more efficient, in the same way the page table is exported to allow user-level memory management.

4.7. Multiprocessor support

As mentioned in Section 4.5., multiprocessor support is not already fully developed in the ExoPC. Basically, the ExOS library, which is linked with the applications to provide the Unix flavor, is mostly not reentrant. This makes it difficult to execute several *cthreads* from the same *environment* (i.e from the same address space) in different processors simultaneously.

ExOS also uses extensively the Xok's synchronous IPC mechanisms. Such mechanism tries to perform a processor handoff, assuming that the target environment is not running. This is not necessarily true in a multiprocessor system, since the target may be running in a different processor.

Nevertheless, the aforementioned limitations are not unsolvable, and Xok-CThreads has been designed and implemented taken into account its use in a multiprocessor environment. Each processor has its own data structures (context queues, etc.) and special care has been taken to optimize parallel operation.

5. Tests

We have developed a prototype of the Xok-CThreads library to check the viability of our proposal to add multithreading features to Xok.

The benchmarks ran on an Intel Pentium-based machine. They 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.

Table 1. Execution times

	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

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. Improving the multiprocessor support

From the experience of implementing a multithreading library and trying to run parallel applications on an exokernel system, we make some proposals for future versions, which could improve the flexibility of the system and its adaptability to multiprocessor systems.

6.1. Xok 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.

From the user-level point of view, few things have changed. When Xok schedules a new *environment*, the current processor information is mapped to a fixed memory address, to transparently handle the current processor.

Somehow, this approach makes the exokernel code simpler, since it behaves as it was running on a single processor machine. Nevertheless, this practically prevents having multiprocessor *environments*, because the multiprocessor information is hidden to them.

So, 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, but to perform processor 'handoffs' to non-running *environments*.

Our proposal consists of breaking the implicit association between an *environment* and an execution flow. An *environment* should be considered simply a resource container, so it may have quanta allocated in different processors and different threads could run in parallel while sharing the *environment*'s resources.

The current semantics of 'blocking' an *environment* (i.e. stopping kernel notifications, such as the prologue executions at quantum start, till certain condition occurs) should be changed into a finer grain blocking. For example, ExOS blocked an *environment* to wait for an I/O operation to finish, but this is not adequate if the *environment* has other threads which can go on working.

In order to support this finer grain management, processor information must be visible from user-level (processor status is not currently available to user-level). Also, the *environment*'s entry points related to scheduling should be specified in a per-processor basis (even in a per-quantum basis). This way, a system library could easily decide what to do at each quantum and at each processor.

Finally, an IPC mechanism should be designed to work efficiently when the target *environment* is running. This can be easily implemented in a multithreading scenario: simply, a user-level IPC handler thread could execute the IPC code in the caller's processor while the main thread goes on executing in its corresponding processor.

6.2. Library support

In order to provide support for a multiprocessor system, it is important to take into account the way libraries behave. In this section we make some proposals to smooth the interaction among different libraries and to adapt them to a multiprocessor system.

First of all, libraries providing services on top of a multiprocessor exokernel system must be reentrant, since a multiprocessor environment can be involved in more than one library call simultaneously.

Libraries in a multiprocessor system can also benefit from using an explicit continuation model [4], instead of a a process-based model, which needs to save and restore a context from the stack. The continuation model reduces the size of the context, and enforces the independence among coexisting libraries.

Finally, the exokernel interface makes it specially suitable to implement libraries based on the scheduler-activation concept [2][6]. The *execution contexts* used for Xok-CThreads are based on this model. *Scheduler activations* have shown to be efficient even in uniprocessor environments [8], 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

The exokernel is a very flexible system. This fact has allowed 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, without a significant overhead.

User-level libraries providing system services should be designed keeping in mind that they will probably have to coexist with other libraries. So, it would be interesting to provide a flexible interface to share data (and possibly kernel events) with other libraries.

As a future work, it is interesting to explore the possibility to apply techniques such as explicit continuations and scheduler activations to exokernel-based systems. This kind of mechanisms can improve the flexibility and efficiency of the user-level system libraries.

We also plan to go on working to increase the multiprocessor support in Xok. This implies increasing the processor information available at user-level. 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). Then, it will be easier to implement programming models different from the classical Unix process model.

8. Acknowledgments

Thanks to Albert Serra, 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.

We also thank the anonymous referees for their comments.

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