CS/TR-94-140

Matching operating systems to application needs: a case study

authors : **Martin Herdieckerhoff and Frédéric Ruget**

project :

state : Approved

classification : Public

distribution :

keywords : CHO,MES

abstract : The increasing complexity of current computer systems has generated new requirements for advanced monitoring and debugging support. This is particularly true for real-time and distributed applications. In this paper, we present new kernel support for monitoring and debugging tools, within the CHORUS micro-kernel architecture. The highlights of the support are its modularity, the use of an upcall mechanism, and a flexible extendable object oriented interface.
This paper also appears in the proceedings of the SIGOPS'94 European Workshop, Dagstuhl, Germany, September 1994.

Matching operating systems to application needs: a case study*†
(Position paper)

Martin Herdieckerhoff
Siemens AG, Dept. ZFE ST SN 1
Otto-Hahn-Ring 6
D-81730, Munich, Germany
mhe@zfe.siemens.de

Frédéric Ruget
Chorus Systems
6, av. Gustave Eiffel
78182 Montigny le Bx, France
ruget@chorus.fr

abstract

The increasing complexity of current computer systems has generated new requirements for advanced monitoring and debugging support. This is particularly true for real-time and distributed applications. In this paper, we present new kernel support for monitoring and debugging tools, within the CHORUS micro-kernel architecture. The highlights of the support are its modularity, the use of an upcall mechanism, and a flexible extendable object oriented interface.

1 Introduction

The increasing complexity of current computer systems has generated new requirements for advanced monitoring and debugging support. This is particularly true for real-time and distributed applications.

In this paper we present new kernel support for monitoring and debugging tools within the CHORUS distributed micro-kernel architecture [Cho92]. This support has enabled us to port two application level tools on CHORUS: PATOC [Her91], a real-time monitor, and CDB [Rug94], a debugger with an execution replay facility for distributed applications running on top of CHORUS. The work is an actual example of a cooperation between a system provider (Chorus Systems) and applications builders (teams from Siemens AG and Alcatel Elin Austria, specialized in monitoring and debugging) to match the operating system to application needs.

During the design phase, we have been faced with several issues as to what kind of monitoring mechanism should be used, what level of monitoring support, and what interface to the monitoring service should be provided. Our approach to the problem has been strongly influenced by the micro-kernel architecture, and our final design is structured by the concept of modularity, the use of an upcall mechanism, and of an object oriented interface.

The remainder of this paper is organized as follows. In Sect. 1.1, we advocate the use of

*This work is partially supported by the European Community under ESPRIT Project 6603 "OUVERTURE".

†This paper also appears in the proceedings of the SIGOPS'94 European Workshop, Dagstuhl, Germany, September 1994.

upcalls as the base mechanism for the monitoring support. In Sect. 1.2, we introduce the object oriented interface to the monitoring service, and we show how it has enabled us to derive more sophisticated, application specific support from the raw micro-kernel support. The global architecture of the monitoring service is sketched in Sect. 1.3. We give a few performance evaluations in Sect. 2. We conclude in Sect. 3.

1.1 An upcall mechanism

The monitoring service is based on a mechanism to notify a client of the occurrence of kernel events. The notification is done via an *upcall* [Cla85] from the micro-kernel to the client of the monitoring service.

Among several other possibilities (e.g. kernel internal counting, event logging by the kernel itself, asynchronous event signalling via IPC, etc.), we think the upcall mechanism is the most appropriate:

- in contrast to a (though more efficient) kernel internal counting scheme, it is much more flexible and puts much less restrictions on the services that may be built on top of the offered kernel primitives.
- in contrast to a kernel internal logging facility [LB90] it prevents the problems related to event buffer management (allocation, periodic spooling, overflow handling).
- its synchronous nature enables the kernel to provide only a minimal number of event parameters at event notification, since the client of the notification can retrieve the other (application specific) parameters by itself (e.g. time, scheduling context...). This makes it an open architecture.

There are however a few drawbacks to using upcalls. For instance, there are some constraints on the kind of operations that the client of the monitoring service can perform while in the context of the upcall. For instance, re-entering the kernel must be limited to a restricted set of system calls. The reason for this restriction is that the kernel is left in a non coherent state during the upcall.

In practice, the set of allowed system calls is the same as in the context of an interrupt handler. For CHORUS, this notably includes the possibility to wake up an ordinary thread (executing in a "standard" context) to perform the work that could not be done directly in the context of the upcall.

A related drawback of using an upcall mechanism is that the client of the monitoring service must be trusted. Indeed, the kernel must have a guarantee that the client will not crash in the context of the upcall, and will only issue allowed system calls. Hence, we have restricted access to the kernel notification service to so called CHORUS *supervisor actors* [1]. As a side effect, this has also enabled us to implement the upcall mechanism very efficiently, because kernel and supervisor actors share the same address space, so there is no address space switch.

[1] A supervisor actor is a special CHORUS actor that shares its address space with the kernel. Supervisor actors always run trusted code, and can be loaded only by a user with super user privileges.

1.2 A C++ interface to the monitoring service

The interface to the monitoring service is object-oriented (it is written in C++ [Str86]). It was designed the following way. We have identified a number of events relevant in the context of kernel monitoring (e.g. creation/deletion of CHORUS objects, thread scheduling events, etc.). To each event we have associated a given CHORUS object, called the object "primarily involved in the event" [HR93] – for instance, the "preemption" event is associated to the "thread" object.

For each CHORUS object (site, actor, thread and port), we have defined a C++ class called the *virtual monitoring class* associated with the object. For each monitoring event that primarily involves the CHORUS object, this class exports a virtual member function called the *virtual monitoring handler* associated with the event. Table 1 describes the virtual monitoring classes associated to the "thread" object [2]. The virtual monitoring classes are made available to clients of the monitoring service by the kernel.

Currently, the specified events are: events of creation/deletion of CHORUS object, hardware events (interrupts, time-outs, traps, exceptions), IPC events, and scheduling events.

The client of the monitoring service must produce (i.e. *derive* in the C++ sense) an implementation for the virtual monitoring classes. For that purpose, it must provide an implentation for the virtual monitoring handlers. These implementation member functions are called *effective monitoring handlers.* They are to be invoked in the client, on occurrence of the associated monitoring event.

The client must then establish a connection with the CHORUS objects that it wants to monitor. This connection is done via a C++ object, called the *interface object.* The interface object must be an instance of the C++ effective class associated with the CHORUS object that is to be monitored. The interface object has two purposes:

- It allows to establish the "kernel-client" connection: via a specific system call, the client provides the kernel with a pointer to the C++ object. The kernel stores this pointer in some internal data structure that represents the CHORUS object to be monitored.
- It is the right place for the client to store state information about the monitored CHORUS object.

Once the connection is established, then the kernel will upcall the appropriate effective monitoring function of the appropriate C++ object, each and every time an event occurs on the connected CHORUS object.

This interface also allows for the implementation of flexible policies regarding the inheritance of the monitoring class. When a new CHORUS object is created by a monitored thread, the kernel invokes a special creation monitoring handler. On invocation of the creation handler, the client of the monitoring service must return either **(1)** a pointer to a new C++ monitoring object that the kernel will associate with the created CHORUS object, or **(2)** a null pointer, thus indicating that the newly created CHORUS object must not be monitored.

[2] The table contains pseudo C++ code: for the sake of clarity, in the function prototypes, we have replaced actual types by symbolic parameter names.

struct KnMonThread : KnMon {	`KnMon` *is reserved for kernel use*
virtual void deleted();	The thread has been deleted.
virtual void disConnected();	Monitoring of the thread has been switched off.
virtual KnMonActor* acCreated(acUi, acKey);	The thread has created an actor.
virtual KnMonThread* thCreated(acUi, acKey, thLi);	The thread has created a thread.
virtual KnMonPort* ptCreated(acUi, acKey, ptLi, ptUi);	The thread has created a port.
virtual void userEvent(evtNo, addr, size);	The thread has issued a user defined event.
virtual void preTrap(trapNo, thCtx);	The thread has made a trap (called before trap handling).
virtual void postTrap(trapNo, thCtx);	The thread has made a trap (called after trap handling).
virtual void preExc(excNo, thCtx);	The thread has made an exception (called before exc. handling).
virtual void postExc(excNo, thCtx);	The thread has made an exception (called after exc. handling).
virtual void run();	The thread has started running.
virtual void preempted(thCtx);	The thread has been preempted.
virtual void beReady(status);	A blocking condition has been removed from the thread's status.
virtual void beUnReady(status);	A blocking condition has been added to the thread's status.
virtual void msgSent(msgLi);	The thread has sent an IPC message.
};	

Table 1: Thread monitoring class exported by the kernel

Note that the client of the monitoring service is not obliged to provide an implementation for all the virtual handlers of a monitoring class. Thus, the monitoring service provides a two-level event filtering:

- First, the kernel notifies the client only of those events that primarily involve a monitored CHORUS object.
- Second, the kernel notifies the client only if it (the client) has provided an effective implementation for the monitoring function associated with the event.

There are benefits to using an object-oriented interface. For instance, it enables to derive enriched interfaces from the "raw" interface. This has enabled us to design a consistent multi-level monitoring architecture, as described in the Sect. 1.3.

1.3 A multi-level monitoring service

The notification service exported by the kernel provides only minimal support. This support does not directly match all clients' requirements. For instance:

- It does not provide for multiple simultaneous connections of differents clients to the same CHORUS object. The kernel keeps at most one pointer to an interface C++ object per CHORUS object.
- It provides only for notifying of events produced by the kernel, although in certain circumstances, it would be useful to be notified of events produced by other layers of the system [3].

[3]This is the case of the "naming" events: to meet the needs of the PATOC and CDB monitors, we have

To compensate for these shortcomings, we have defined an additional EXTMON [4] module, which is capable of managing **(1)** the multiplexing of event notifications to several simultaneously connected clients and **(2)** the notification of events produced by other layers of the system. This module interposes between the micro-kernel and the clients of the notification service. To the micro-kernel, it appears as an ordinary client of the "raw" notification service.

It was possible to derive (in the C++ sense) the interface to the notification service exported by EXTMON from the raw interface exported by the kernel. This is illustrated by Tab. 2 and 3, which describe the monitoring classes exported by the micro-kernel and by EXTMON for the "site" object.

Obviously, the functionalities provided by EXTMON could have been directly implemented at the micro-kernel level. However, implementing it in a separate module implies several advantages:

- The EXTMON module is optional. It is possible to get rid of it when it is necessary to a light weight configuration of the kernel (Cf. Tab. 4), or to have the most efficient notification service possible.
- It is possible to have several different versions of the EXTMON module, each implementing a specific policy, regarding the multiplexing of event notifications (which client must be notified first - should all clients be notified) or the naming service (should symbolic names be unique). Using an external module allows not to make a choice at the micro-kernel level.

Thus the general architecture of the notification service is that described by Fig. 1. When an event occurs that primarily involves a monitored CHORUS object, the event is first reported to the EXTMON module [5]. In turn the EXTMON module dispatches the notification to all local (supervisor) monitors that are connected to the CHORUS object. Finally, the local monitor may use CHORUS IPC to communicate with central monitors in user space.

2 Evaluation

The impact of the notification service on the size of the kernel is illustrated by Tab. 4. The overhead on kernel code size is less than 1%.

We have run the GAEDE macro benchmarks which show that the impact of the notification service on macroscopic computations of the system is negligible (we do not have enough space to show the results here).

The impact of the notification service on microscopic performance is illustrated by Tab. 5, which shows results obtained with the CHORUS KBENCH micro benchmarks. The overhead due to enabling the notification service without actually using it is negligible. The overhead due to monitoring CHORUS objects through the "raw" kernel or EXTMON interfaces is approximately 20%. The cause of this overhead is the fact that the kernel

developed a symbolic (ASCII) name server for CHORUS objects. This service is implemented as a separated actor running on top of the micro-kernel. Thus the naming events cannot directly by reported by the micro-kernel's notification service. However, the PATOC monitor requires some kind of notification support, even for naming events.

[4] EXTMON stands for *EXTended MONitoring.*

[5] In fact, some events are directly created at the EXTMON level (e.g. naming events).

struct KnMonSite : KnMon {	`KnMon` *is reserved for kernel use*
virtual void disConnected();	Monitoring of the site has been switched off.
virtual void intrBegin(intrNo);	An interrupt has occurred on the site (called before intr. handling).
virtual void intrEnd(intrNo);	An interrupt has occurred on the site (called after intr. handling).
};	

Table 2: Site monitoring class exported by the kernel

struct MonSite : KnMonSite {	
dlink monLink;	*Reserved for kernel use (multiplexing).*
virtual void siteNamed(name);	The site has been renamed.
virtual void acNamed(acUi, acKey, name);	An actor of the site has been renamed.
virtual void thNamed(acUi, acKey, thLi, name);	A thread of the site has been renamed.
virtual void ptNamed(acUi, acKey, ptLi, name);	A port of the site has been renamed.
};	

Table 3: Site monitoring class exported by EXTMON

must prepare the parameters for the monitoring handlers.

3 Conclusion

We have extended the CHORUS micro-kernel with kernel support destined for tools that monitor the actions occurring on kernel objects. The highlights of this support are its modularity, the use an upcall mechanism for the notification of events, and a flexible extendable object oriented interface. It should meet the needs of a variety of applicative tools while preserving the necessary efficiency and generality of the micro-kernel. It has enabled us to port the PATOC [6] and CDB [7] monitoring and debugging tools on top of the CHORUS microkernel.

[6] As of July 94, porting of PATOC is underway.

[7] A first prototype of CDB with the complete re-execution facility has already been implemented.

References

[Cho92] Chorus Team. Overview of the Chorus distributed operating system. In *USENIX Workshop on Micro Kernels and Other Kernel Architecthres*, Seattle (USA), 1992.

[Cla85] D. D. Clark. The structuring of systems using upcalls. In *SOSP10*, pages 171–180, Orcas Island, WA, December 1985.

[Her91] M. Herdieckerhoff. Implementation of PATOC on the EDS testbed. Technical Report EDS.DD.8F.0027, ESPRIT II, January 1991.

[HR93] M. Herdieckerhoff and F. Ruget. CHORUS monitoring hooks specifications and manual pages. Technical Report OU/TR-93-23, Chorus Systems, 1993.

[LB90] T. Lehr and D. L. Black. Mach kernel monitor (with applications using the pie environment).
Available on host mach.cs.cmu.edu in /usr/mach/public/doc/-unpublished/monmanual.ps through anonymous FTP, February 1990.

[Rug94] F. Ruget. A distributed execution replay facility for CHORUS. In *Proc. of*

	text	data	bss	total
std	258516	78278	87242	424036
m	263876	79350	87242	430468
due to notification service	+1612	+928	+0	+2540
due to misc. services	+3748	+144	+0	+3892
EXTMON module	12156	2924	20292	35372

std: a standard, rather big configuration of the CHORUS v3 r4 micro-kernel, including full kernel debugger, full IPC support, full distributed virtual memory support.
m: a configuration derived from std by adding the monitoring support.

Table 4: Kernel sizes

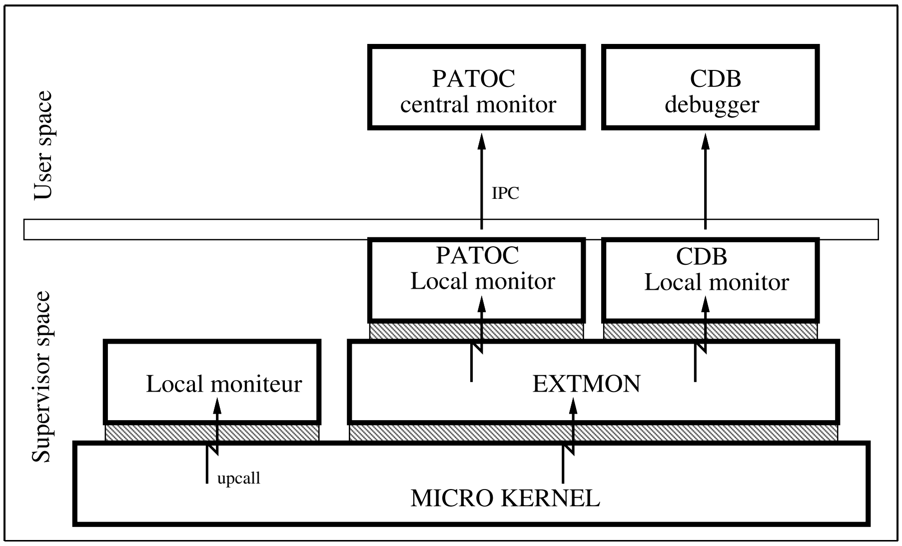

Figure 1: Global architecture of the notification service

the 7th Int. Conf. on Parallel and Distributed Systems (PDCS'94), Las Vegas, Nevada, October 1994.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 1986.

	std	m	md	mdd
`threadSelf()` system call	2.1 ± 0.5	1.1 ± 0.6	1.1 ± 0.3	2.0 ± 0.4
`actorSelf()` system call	3.1 ± 0.6	2.2 ± 0.6	3.4 ± 0.6	3.2 ± 0.6
context switch	24.8 ± 0.8	27.9 ± 0.7	28.3 ± 1.0	33.2 ± 1.3
interrupt handler entry	14.1 ± 0.4	15.9 ± 0.0	21.4 ± 0.5	18.2 ± 0.8
interrupt handler return	11.6 ± 0.4	12.3 ± 0.5	15.3 ± 0.6	15.5 ± 0.3
trap handler entry	4.8 ± 0.4	3.5 ± 0.4	5.9 ± 0.7	4.4 ± 0.3
trap handler return	2.0 ± 0.3	1.8 ± 0.3	2.0 ± 0.3	2.0 ± 0.3
actor creation	76.1 ± 0.7	82.7 ± 0.6	88.0 ± 0.6	164.3 ± 4.0
thread creation	91.0 ± 0.5	91.6 ± 0.6	101.4 ± 0.4	119.6 ± 0.9
IPC send (body 4kb)	443.3 ± 1.3	448.2 ± 1.0	462.7 ± 1.1	466.7 ± 1.0
IPC receive (body 4kb)	455.5 ± 1.7	453.3 ± 1.0	493.6 ± 1.3	510.6 ± 1.0

std: a standard configuration of the micro-kernel.
m: monitoring support enabled. No kernel object is actually monitored.
md: all kernel objects are monitored via the "raw" interface.
mdd: all kernel objects are monitored via the EXTMON interface.

Table 5: Cost of the notification service (μs)