CS/TR-91-97

Give a Process to your Drivers!

François Armand

---

1. Abstract

This paper presents how the modular architecture used for the CHORUS/MiX V.3.2 system enables system writers to encapsulate UNIX ® device drivers within UNIX processes, and the possibilities offered by this feature.

The CHORUS ® architecture is designed to support a new generation of open and distributed operating systems. A microkernel provides generic services allowing servers that cooperate within subsystems to offer standard interfaces: a UNIX SVR3.2 interface is available and a UNIX SVR4 interface is under development. The CHORUS microkernel provides services to manipulate actors and threads (memory management, scheduling...). It also offers a distributed Inter-Process Communication (IPC) facility and services to dynamically connect user provided functions to hardware interrupts and traps.
CHORUS/MiX V.3.2 is implemented as a set of the following servers: a Process Manager, a File Manager, a Terminal Manager and a Socket Manager (to provide BSD sockets). In addition to standard UNIX services (which have been transparently extended to distribution), one can also use CHORUS services to take advantage of CHORUS real-time features and multi-threaded processes.

The CHORUS/MiX V.3.2 has been experimentally enriched by the introduction of a new kind of server named a "Driver Actor" (DA). A DA allows one to encapsulate device drivers within independent actors. This new kind of server has been (experimentally) used to separate the disk driver from the File Manager.
As a consequence, this separation removes the constraint placed on the File Manager to reside

---

In: Proceedings of the EurOpen Autumn 1991 Conference, Budapest, Hungary, September 16-20, 1991

® UNIX is a registered trademark of USL.

® CHORUS is a registered trademark of Chorus systèmes.

into the machine supervisor address space, as priviledged instructions are executed only by, the driver itself. Thus, the File Manager can run as a UNIX process either in user or supervisor address space, communicating with the driver through CHORUS IPC accessible at UNIX interface level. This allows one to take advantage of standard debuggers such as sdb, gdb, to debug the File Manager.

Using CHORUS IPC between the File Manager and the Driver Actor permits them to be transparently distributed over a network of processors. Therefore the driver may be loaded on a dedicated board while providing this driver with an environment compatible with that of a Unix native kernel. This kind of configuration is used within the MultiWorks Esprit project.

In addition, the CHORUS/MiX subsystem permits one to dynamically load processes running in the supervisor address space usually reserved for use only by the UNIX kernel. Thus from a shell, one can dynamically load, locally or remotely, UNIX drivers as if they were "common" processes.

2. CHORUS/MiX

2.1 CHORUS Architecture

A CHORUS System is composed of a small-sized **Nucleus** and of possibly several **System Servers** that cooperate in the context of **subsystems** to provide a coherent set of services and user interface. A detailed description of the CHORUS system can be found in[Rozier88a] Some other systems have adopted similar architectures: Mach [Accett86a] ,V-system[Cherit88a] and Amoeba[Mullena]

2.2 CHORUS Kernel

The CHORUS kernel provides the following basic abstractions:

- The **actor** defines an address space that can be either in user space or in supervisor space. In the later case, the actor has access to the priviledged execution mode of the hardware. User actors have a protected address space.
- One or more **threads** (*light weight processes*) can run simultaneously within the same actor. They can communicate using the memory of the actor if they run in the same actor.
- Otherwise, they can communicate through the CHORUS IPC that enables them to exchange **messages** through **ports** designed by global unique identifiers.
  A message is composed of a (optional) **body** and an (optional) **annex**. Annex size is fixed (64 bytes currently). Body size is variable.

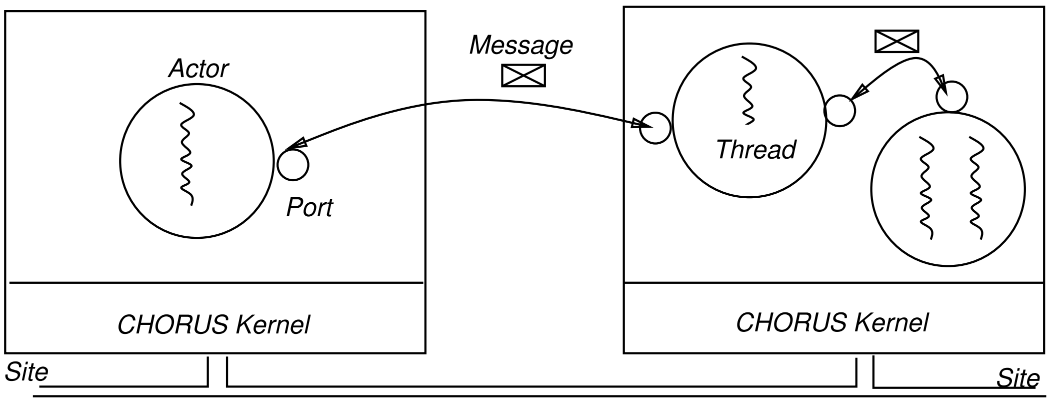

Figure 1. – CHORUS basic abstractions

2.3 The MiX subsystem

MiX is a CHORUS subsystem providing a UNIX interface that is compatible with UNIX SVR3.2. It is binary comptible with SCO on AT/386 machines. The paper [Herrma88a] provides more details on the implementation of the MiX subsystem. It is composed of the following servers:

- The **Process Manager** (**P.M.**) provides the UNIX interface to processes. It implements services for process management such as the creation and destruction of processes or the sending of signals. It manages the system context of each process that runs on its site. When the P.M. is not able to serve a UNIX system call by itself, it calls other servers, as appropriate, using CHORUS IPC.
- The **File Manager** (**F.M.**) performs file management services.
- The **Device Manager** (**D.M.**) manages asynchronous lines, keyboards, pseudo-ttys, etc and implements the UNIX line disciplines.
- The **Socket Manager** (**S.M.**) implements BSD 4.3 socket services, providing access to TCP/IP protocols.

For performance reasons and because they manage traps or interrupts these servers run in system space.

UNIX processes are implemented by the P.M. on top of the abstractions provided by the CHORUS kernel. Basically, a UNIX process is composed of one actor within which one thread is running.

2.4 Extensions

One goal of CHORUS/MiX is to offer the user new services:

- providing multithreaded processes,
- transparently extending traditional UNIX services to distribution, thus providing a distributed file system, remote execution of processes and distributed signals,

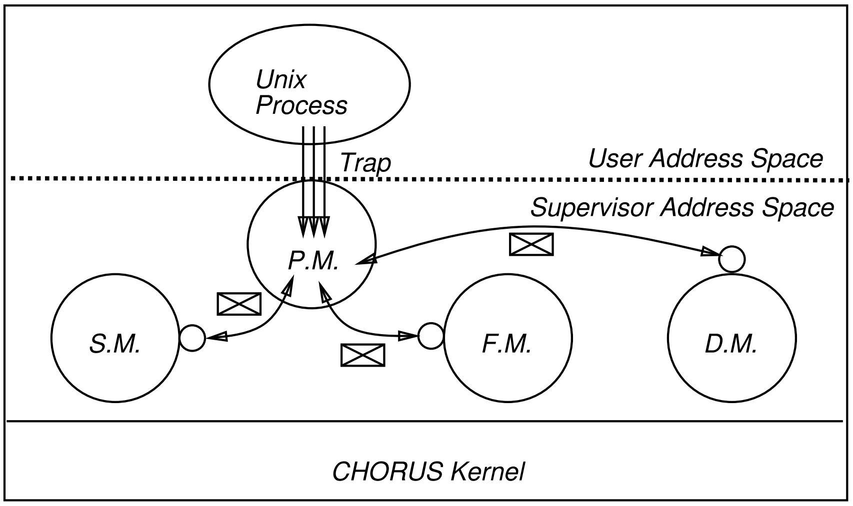

Figure 2. – MiX subsystem structure

- providing CHORUS IPC at the UNIX interface level,
- providing access to the priority-based preemptive scheduling,
- providing the ability to dynamically execute processes in system space.

More details on these extensions can be found in:[Armand89a] and[Armand90a] The later possibility is extremely important to the rest of this paper. The system space is partitioned between the CHORUS kernel itself and CHORUS/MiX servers. The free space is available for the user to load processes dynamically in system space. Loading processes into system space is desirable to achieve better performance or gain access to the priviledged mode of execution of the hardware. This ability is restricted to the UNIX super-user.

3. History and Related Works

3.1 CHORUS/MiX and the drivers

CHORUS/MiX has always tried emphasise on modularity by splitting the main UNIX kernel functions into independent modules. In the current CHORUS/MiX V.3.2, however, one basic service has not been "separated" in such a way: device drivers:

- Drivers such as disks, floppies and tapes, offering a "block interface" are embedded within the File Manager.
- Drivers for terminals, keyboard/mouse offering only a "character interface" are included in the Device Manager.

It was deemed more important to separate file management from process management and, thus, to be able to distribute them over a network, than to separate a disk driver from the file management as the management of files is tightly coupled with the management of a disk.

However, CHORUS/MiX has already experimented with partial driver separation in its previous versions:

- CHORUS V2

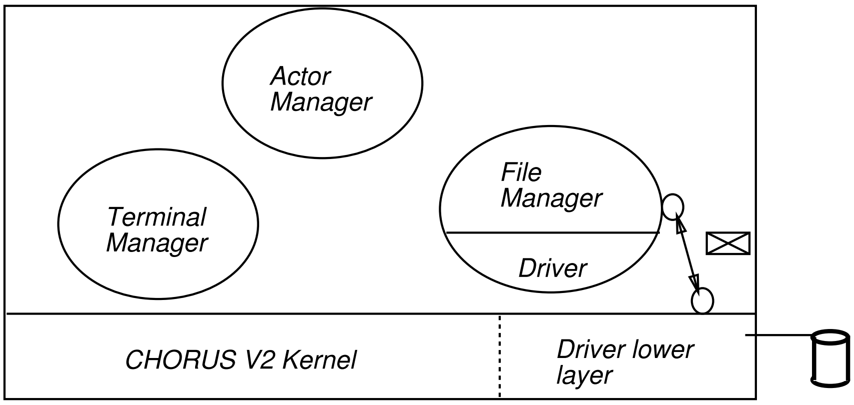

Figure 3. – I/O Management in CHORUS V2

In version V2, UNIX servers ran as "user mode" actors, and thus had no access to priviledged hardware instructions. I/O operations were performed by the kernel upon reception of request messages sent by the UNIX servers. As it was a very low-level interface, UNIX drivers had to be modified. Moreover, this interface was too dependant on the machine on which the system ran.

- CHORUS V3.0
  In the first implementation of CHORUS/MiX on Bull SPS 7/70, the File Manager executed in user mode. The sequences of instructions of the disk driver that contained priviledged instructions were separated in routines running in system mode that were dynamically connected to a trap, during driver initialization. Once again, this approach involved a specific adaptation for each driver being ported.

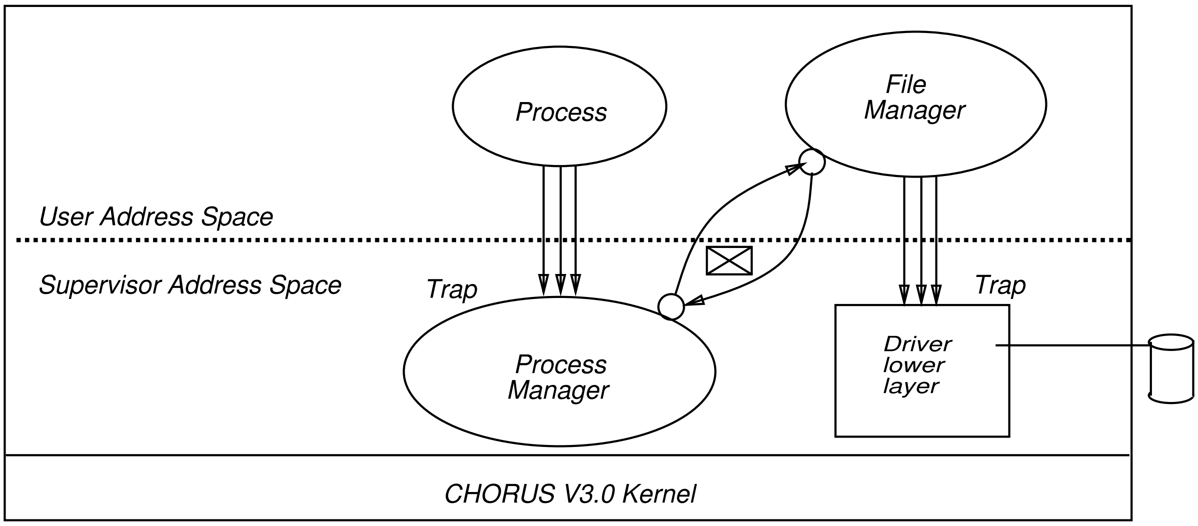

Figure 4. – Previous I/O Management in CHORUS V3.0

None of these two attempts fit with the "natural" driver interface provided in UNIX kernels.

3.2 Related Works

- MACH:
  MACH 3.0[Golub90a] also provides a UNIX implementation based on a microkernel. The UNIX emulation runs in user mode within a "single" server communicating with the drivers that are linked together with the microkernel. Communication is achieved using MACH IPC. This is similar to CHORUS V2 with the important distinction that the interface used in MACH is more natural. However, there is no provision for the dynamic dynamic loading and unloading of device drivers as drivers are embedded within the microkernel.

- V System:
  In V[Cherit90a] device drivers are embedded within a pseudo process that executes in the V kernel itself. Access to the devices is achieved by means of a UIO (Uniform Input/Output) interface mapped on the message based V IPC. The UIO interface is generic enough to allow access to any kind of servers, such as file servers, mail servers or device servers, in a quite uniform fashion. More details about the UIO mechanism can be found in[Cherit84a] An application gets access to the file server using the UIO interface, possibly hidden within a library. In turn, the file server accesses the device driver using the same interface. However, as in MACH, it seems that there is no provision for dynamically loading a device driver.

- SunOS:
  On a Sun i386, SunOS allows one to dynamically load a driver within the kernel using a special command "*modload*". But the driver's writer needs to provide some "wrapper" code in addition to the driver itself. To our knowledge this is the system that achieves the functionnalities that are the most similar to CHORUS/MiX. Unfortunately, this ability is restricted to Sun i386 machines.

3.3 Why do we need Driver Actor ?

3.3.1 Powerful Device Boards: the MultiWorks example

More and more machines provide powerful boards (processor and plenty of memory) to connect devices. The European project MultiWorks builds such a multi-media workstation.

The station is built around an EISA bus with a Main Processing Unit, based on a Intel CP486, and several boards, called Intelligent Peripheral Adaptators. I.P.A. are dedicated to device management and based on the chip Acorn ARM3. An I.P.A. board has 1, 4 or 16 megabytes of RAM memory. One of these board is used to manage a "Disk Array" through a SCSI bus.

The CHORUS kernel runs on both the CP486 and I.P.A. providing IPC over the EISA bus. In addition, a full CHORUS/MiX subsystem runs on the CP486 board. Thus, CHORUS/MiX has to manage a disk which is connected to an IPA. One could have taken advantage of the CHORUS/MiX modularity by running the File Manager on the I.P.A. But, this solution has a major drawback: a lot of UNIX requests that do not require a disk access, such as operations on pipes or on cached data, would generate undesirable accesses to the EISA bus. The additional bus references would significantly reduce the available bus bandwidth and would negatively impact the performance of both the system and the applications.

It seems more appropriate to execute only the disk driver on the I.P.A. and to let the File Manager and its buffer cache reside on the main processor.

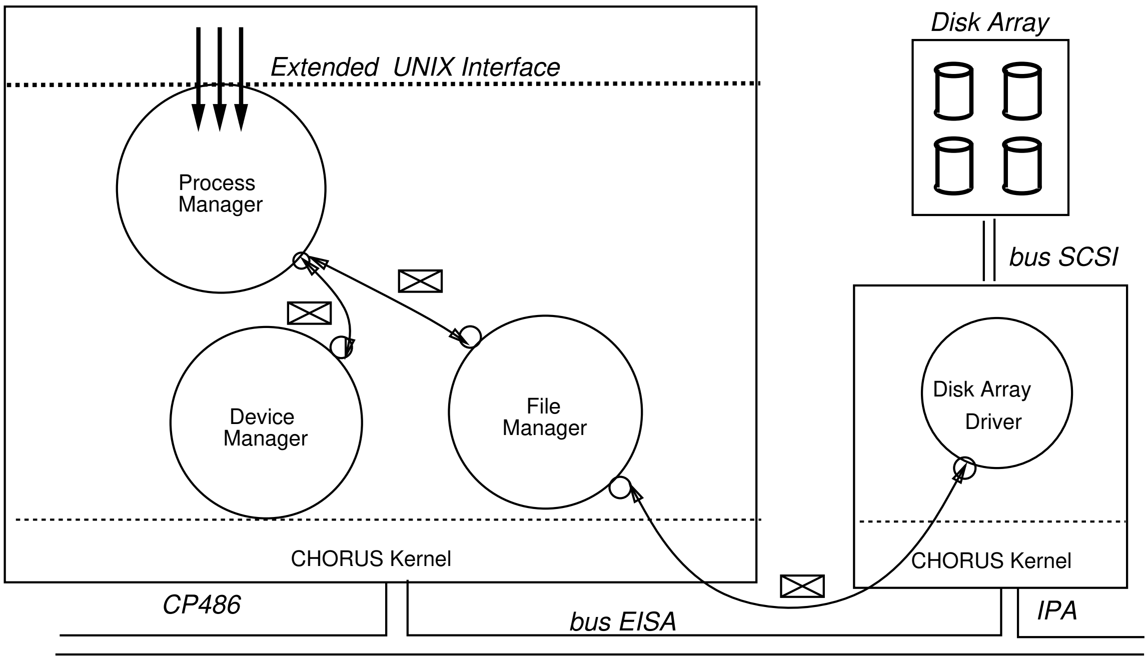

Figure 5. – MultiWorks Multimedia Workstation Architecture

3.3.2 Co-Existing sub-systems

The CHORUS kernel allows several subsystems to run simultaneoulsy on a same machine. In such a case, various subsystems share the access to the processor and to the main memory, relying on the services provided by the CHORUS kernel. Access to other physical resources, such as devices, is not managed by the kernel, but is on the responsibility of the subsystems themselves.

Rather than partitionning the devices between the subsystems, it is more convenient to be able to share the access of a device between multiple subsystems. To achieve this, the device driver must be able to communicate with each of the subsystems. Splitting the drivers from the UNIX servers that need them, is a first step in that direction.

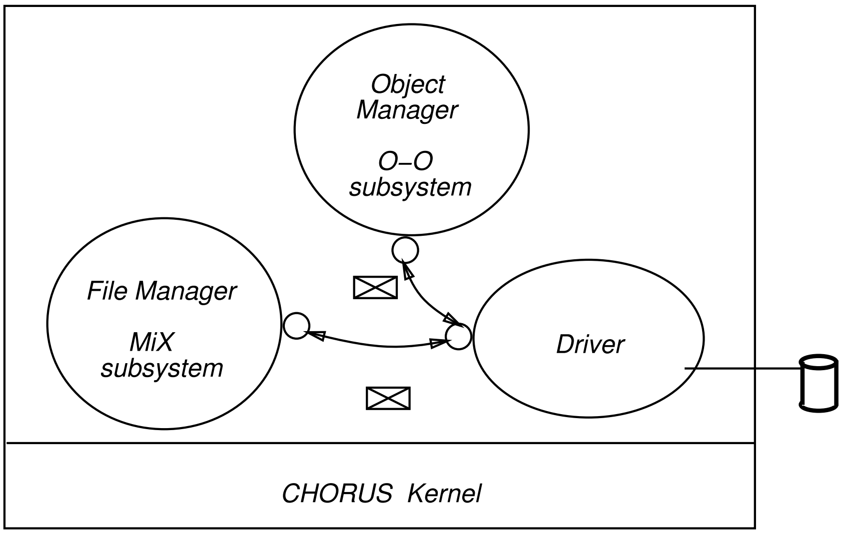

Figure 6. – Sharing a Driver between two subsystems

3.3.3 Other Needs

The encapsulation of drivers within independent servers presents some other interesting characteristics:

- The combination of this encapsulation with the ability to dynamically load programs in system space enables us to dynamically load drivers without stopping the system when adding a new device type to the machine. This can also help to reduce the size of the resident set of system servers by removing drivers that are not frequently used from the default boot file. These drivers being loaded and unloaded on demand either manually or automatically.
- Servers such as the File Manager, being "cleansed" of priviledged hardware instructions can be executed in user mode with a loss in performance the. This configuration can be used to debug the CHORUS/MiX servers as "normal" UNIX processes, using standard debuggers such as sdb dbx, or gdb.

4. General Structure of an Actor Driver

4.1 Exported Services

Services provided by a driver are well-defined within UNIX kernels, although this definition may vary from one system to another. Nevertheless, basic services remain the same, they are summarized below.

UNIX distinguihes between drivers that offer a "*block interface*" and those that offer a "*character interface*".

- "**Block Interface**"
  Such a driver has to provide the following services:
  — *drv*open (device, read/write, open_type)
  — *drv*close (device, read/write, open_type)
  — *drv*strategy (buffer_header)
  The most important function is the one called "*strategy*" which is used to process the I/O's on the device. The communication between the upper layers of the file system and the driver is achieved through a "*buffer header*". The useful information for the driver is contained in the following fields of such a header:
  — Device number on which the I/O must be performed,
  — Block number to be read/written,
  — Size of the I/O,
  — Address in memory to store/find the data,
  — Flags to describe the I/O (read/write, synchronous/asynchronous, etc)

- "**Character Interface**"
  Such a driver must export the following services:
  — *drv*open (device, read/write),
  — *drv*close (device, read/write),
  — *drv*read (device)
  — *drv*write (device)
  — *drv*ioctl (device, cmd, arg)
  The description of the I/O to be performed is defined in a "*uio*" structure within the system context of the process.

4.2 Imported Services

Unfortunately, no standard or guide defines the services provided by a UNIX SVR3.2 kernel that can be used by a device driver. Thus, a driver can potentially use all of the kernel functions and data structures. However, in practice device drivers use a small set of services and data structures[1]. It is, thus, possible to list the most common needs of a driver:

- sleep/wakeup: wait for event, reactivate a process when the event occurs,
- spl0, spl7, splx, etc: mask / unmas interrupts,
- timeout, untimeout: activate a function after a delay, cancel the activation,

1. UNIX SVR4 defines precisely what can be used by a device driver within a "DDI/DKI Reference Manual".

- getblk, geteblk: allocate a buffer and a buffer header, etc,
- biowait, biodone: wait for the completion of an I/O, indicate the completion of an I/O.

All these services have already been emulated within CHORUS/MiX servers. Thus, it is straightforward to re-use such functions within a Driver Actor.

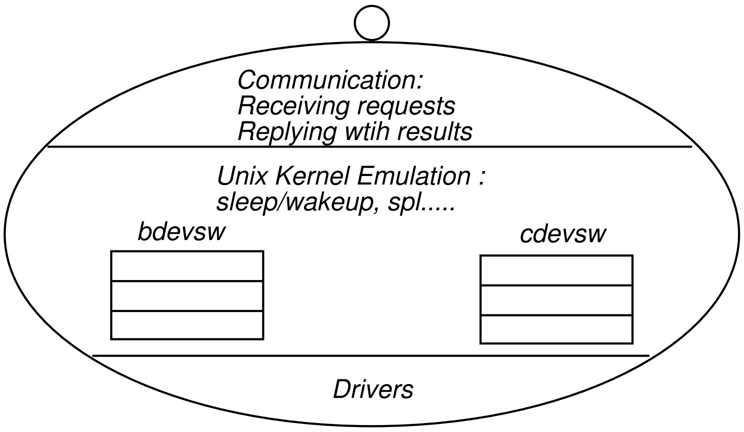

Figure 7. – Driver Actor Structure

4.3 Layout and Communication

A Driver Actor is similar to other servers in that it is a multi-threaded server that receives requests on a port. As the real device driver is no longer embedded within the server, it is necessary to replace it by a "stub-driver" that conforms to the UNIX driver interface and that sets up request messages, sends them to the appropriate Driver Actor, and waits for the corresponding reply. This structure implies that the stub-driver is able to retrieve the UI of the port of the Driver Actor given a major number.

Upon reception of a request, the Driver Actor needs to unmarshall the message and to build a context similar to that which would have existed within a monolithic UNIX kernel so that the appropriate driver function can be left unmodified. Upon completion of the request, the Driver Actor must marshall the results in a reply message that will be interpreted by the stub-driver.

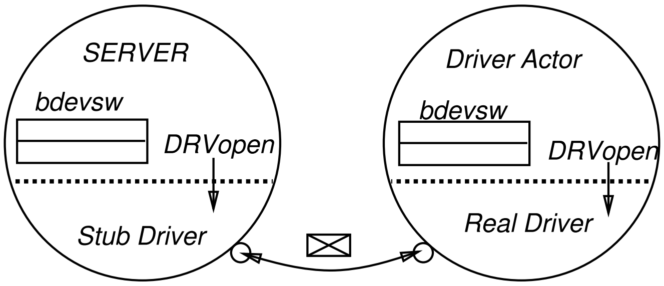

Figure 8. – Communication Server <-> Driver Actor

5. A Sample Case: The Disk Driver Actor

5.1 Description

This architecture has been used within CHORUS/MiX V.3.2 to separate the disk driver from the File Manager. The current implementation deals only with the block interface, the character interface of the diskdriver will be implemented later. The contents of the messages being exchanged derive directly from the UNIX driver interface described previously.

The "write" request passed to the stub-driver by the File Manager is in fact, according to the block interface, a "strategy" request with a write flag. The stub-driver receives a buffer header from which it extracts the information needed by the real driver and copies it into a message annex. Data to be written to the disk are simultaneously transmitted within the message body.

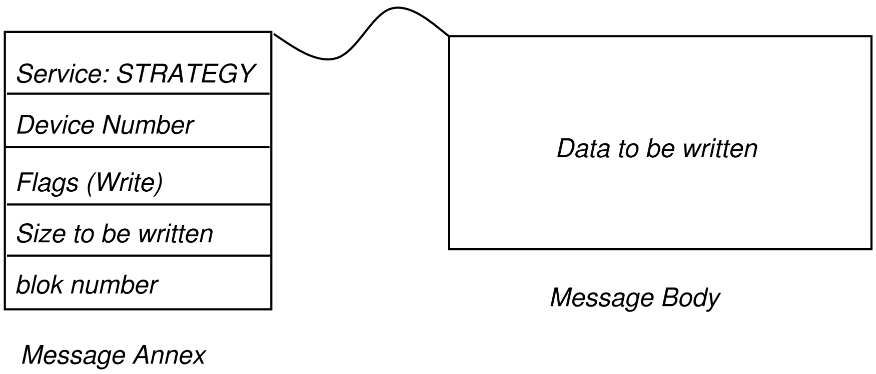

Figure 9. – Disk Write Message Request

Thus, a synchronous write request is performed as follows:

- The *strategy* routine of the stub-driver is invoked by the File Manager. It receives a buffer header describing the I/O to be performed and pointing to the buffer containing the data to be written.
- The *stub-strategy* routine sets up a message annex with the information contained in the buffer header. The message body is the buffer itself. No copy of data is performed by the stub-driver. Then, the stub driver invokes the real driver by means of the remote procedure call offered by the CHORUS kernel.
- The CHORUS IPC Manager will perform the copy of the message body and will make the message available to the Driver Actor.
- The Driver Actor receives the message and decodes the service code found at the beginning of the annex. Its only work is to set up a request context emulating a UNIX kernel environment, and to set up a buffer header from the information received in the message annex. The pointer to the buffer containing the data is set to the body of the received message.
- Then, the real strategy routine is invoked. The Driver Actor then waits for the I/O completion, using *biowait*. When the I/O is completed, the real driver will wake up the request, using *biodone*, permitting the Driver Actor to reply to the File Manager.

Read requests are processed in a similar fashion.

Two important issues need more explanation:

- **Data Transfer**
  When the Driver Actor and the File Manager are on the same site, the modularity implies that an additional copy has to be performed. Of course, that copy is not necessary when the driver is running within the File Manager. The UNIX kernel algorithms guarantee that while an I/O is being processed, the buffer cannot be accessed by any other process. One could take advantage of this "property" to attribute the buffer to the driver for the duration of the I/O.. Unfortunately, on AT/386 machines the page size is 4Kbytes and System V buffers are 1Kbytes long. Thus, it would be necessary either to place only one buffer in a page, wasting a lot of physical memory, or to lock four buffers at a time implying more complex synchronization and more contention in the File Manager. With file systems using 4Kbytes buffer, such as B.S.D. or SVR4, this could be implemented.
- **Synchronous and Asynchronous processing**
  In a UNIX kernel as well as in the CHORUS/MiX File Manager, an I/O request can be processed either synchronously or asynchronously.
  — Synchronous requests
  Theses kinds of requests are started by the invocation of the *strategy* routine which returns immediately. The completion of the I/O is awaited using the *biowait* function, putting the current process to sleep. Thus the duration of the I/O can be used to give the main processor to any runnable process. This behaviour is emulated by releasing the processor in the File Manager when invoking the Actor Driver. When the reply is received, the processor is re-acquired, and the stub-driver returns from its strategy routine. The following call to *biowait* that is performed later by the File Manger will just check that the I/O is done without blocking the process.
  — Asynchronous requests
  One of the characteristics of UNIX file systems is their use of asynchronous I/O operations such as deferred writes and read ahead to get better performance. In such cases, the

I/O is started as usual by invocation of the strategy routine but as its result is not necessary to the current process, its completion is not awaited . The completion of the I/O is just marked within the buffer header.

The protocol between the File Manager and the Driver Actor described previously has been modified to satisfy this need. Upon reception of an asynchronous request the Driver Actor answers immediately, enabling the File Manager to continue its work. When the I/O is done, the Driver Actor sends an asynchronous message to the File Manager denoting that the I/O has completed. In addition, a completed read request carries the data. In essence, this mechanism is similar to that of a software interrupt. In order to process these asynchronous messages from the Driver Actor, the stub-driver needs to create a thread that will process them.

5.2 Usage: Dynamic Reconfiguration

In order to validate the above design, we have imagined and succesfully demonstrated a scenario in which the hardware configuration evolves. Thus, it is up to the system software to follow the evolution!
The initial configuration is composed of one autonomous machine with a processor,some memory, a disk, some terminals and a network interface. Another machine has the same configuration except that it has no disk. The first machine runs a full CHORUS/MiX system consisting of the Kernel, PM, FM, DM and SM. On the diskless machine a CHORUS/MiX system without File Manager is loaded. For demonstrational purposes, the two machines are COMPAQ 386; the disk of one of them being not used. The distribution provided by CHORUS enables one to use the diskless machine as a second processor, and thus to balance the load between the two processors.

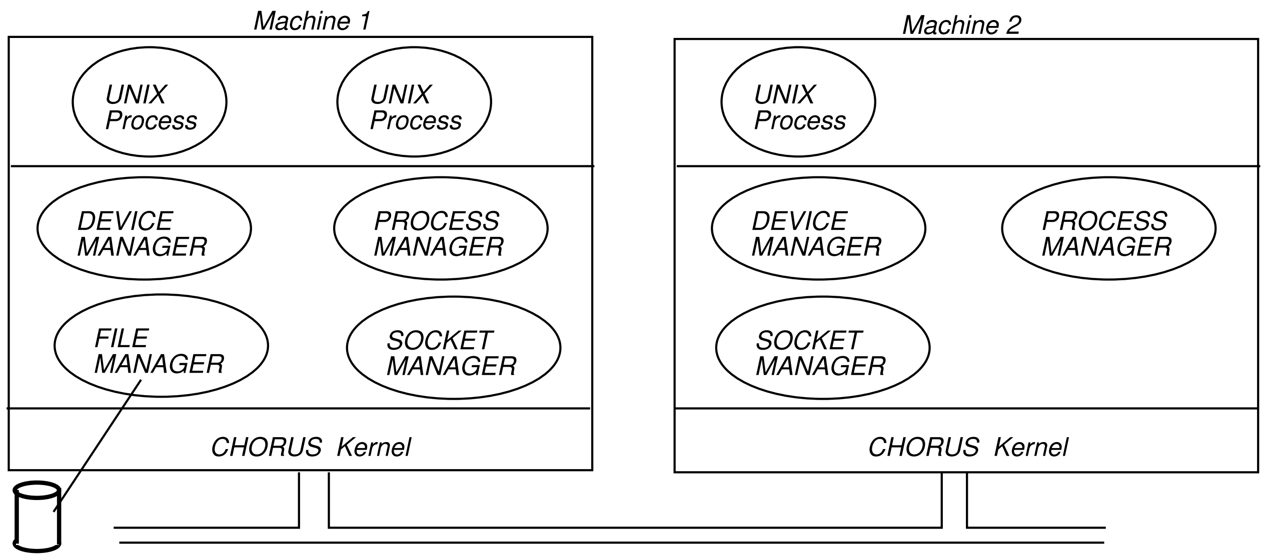

Figure 10. – Using a diskless machine

Next, the scenario script states that as the user's needs are increasing the diskless machine is equipped with a local disk. One supposes the hardware "smart" enough to allow this addition wihtout stopping the system. Thus, one can remotely load a disk driver, from the first machine, within the supervisor address space of the second machine. This allows one to initialize the new disk and to copy on it the files needed by an autonomous machine.

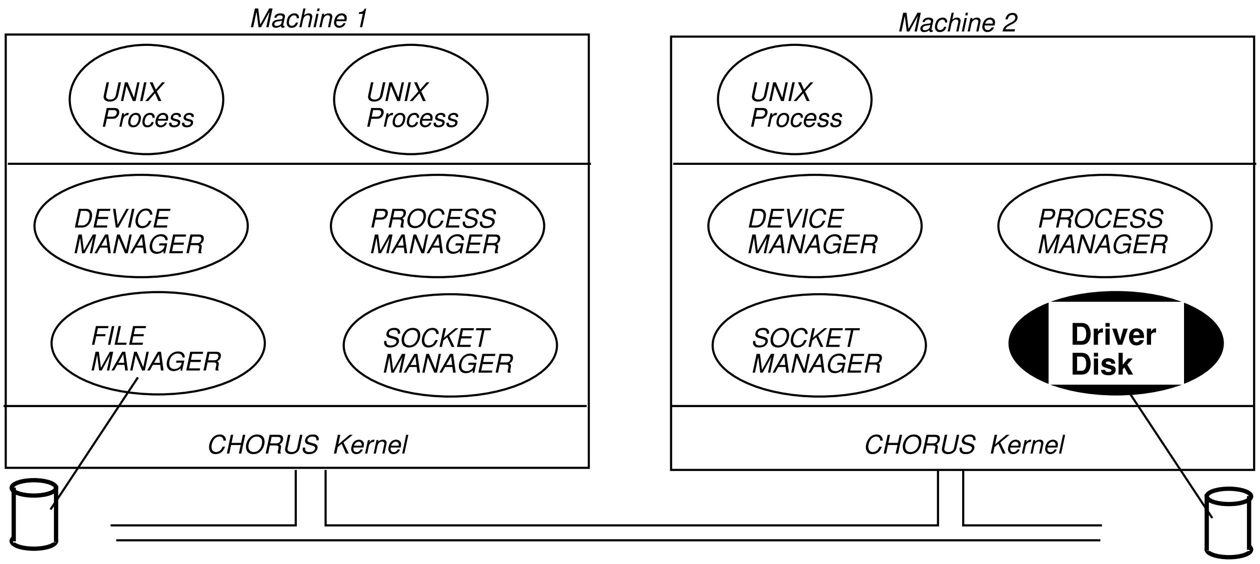

Figure 11. – Dynamically Loading a Disk Driver

Finally, once the new disk has been initialized, one can remotely load on the second machine a File Manager running as a UNIX user process. The Process Manager will recognize the existence of a local File Manager and will start the execution of the "/etc/init" process loaded from the new disk. The second machine will then be completely autonomous.

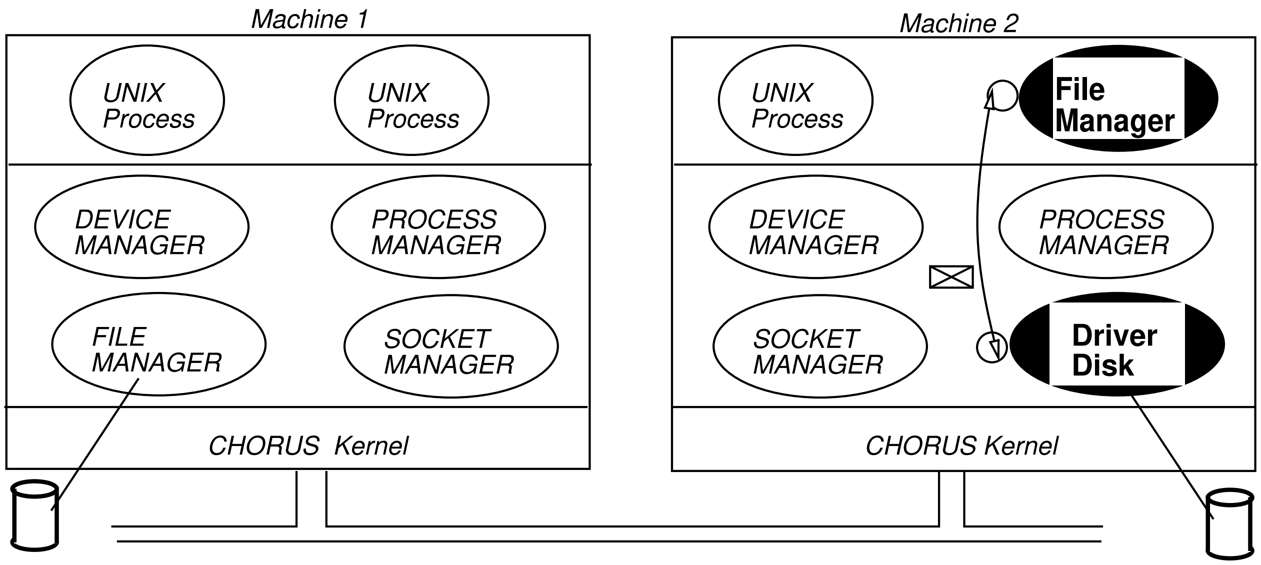

Figure 12. – Dynamically Loading a File Manager

As the File Manager is being executed as a "normal" UNIX process, one can use UNIX debuggers to debug it, although some "obvious" precautions must be taken. When a process being debugged encounters a "breakpoint" it is stopped by the system. When the process being stopped is the File Manager, the requests it receives will not be processed until its execution will be resumed by the debugger. Note that some of these requests will originate from the debugger itself, if its is running locally. Thus, it is necessary to make the File Manager sources available on another machine and to run the debugger from another machine to avoid deadlock.

5.3 Performances

In order to appreciate the effect of such an architecture on the system's performance we have run a set of benchmarks in various configurations:

- standard CHORUS/MiX with the File Manager and the disk driver linked together within one actor.
- File Manager running as UNIX process in user mode, and the disk driver running as a UNIX process in supervisor mode (as in the demonstration presented above).
- File Manager and driver disk both running as UNIX processes in supervisor mode.

The benchmarks run are briefly described:

- **creat**: mesures the time needed for the pair *creat*(2) / *close*(2).
- **open**: mesures the time consumed by the pair *open*(2) / *close*(2).
- **write1k**: mesures the time consumed to write 2 megabytes in a regular file 1 kilobyte at a time.

- **read1k**: mesures the time consumed to read 2 megabytes from a regular file 1 kilobyte at a time.

These benchmarks have been run on a COMPAQ386 running at 20 MHz with 5 Megabytes of main memory.

	"Standard" MiX	FM as a user process	FM as a supervisor process
creat	60 creat/sec	58 creat/sec	59 creat/sec
-			
open	443 open/sec	355 open/sec	428 open/sec
-			
write1k	60 Kbytes/sec	43 Kbytes/sec	49 Kbytes/sec
-			
read1k	198 Kbytes/sec	90 Kbytes/sec	110 Kbytes/sec

The case where the File Manager runs in user space is, of course, slower because data must be copied from the benchmark process to the File Manager, which is also a user process. Such a copy implies a memory context switch which is unnecessary when the copy is done from user space to supervisor space.

The stability of the "*creat*" test is probably due to the fact file creation implies synchronous writes to the disk to write the inode. Thus, this test is mostly limited by the disk transfer rate and not by the software algorithms.

The "*open*" test consist essentially in copying a pathname between the user process and the File Manager. The user mode FM is slower because of the extra memory context switch. The two other configurations where the FM executes in supervisor space achieve similar performance. The situation where the disk driver runs outside of the FM is penalized by the first open when it is necessary to load blocks from the disk.

The most important differences are observed on read/write operations. The difference between the standard MiX and the case "FM as a supervisor process" with a separated driver disk is due to two main reasons:

— First, an extra copy is performed for all disk blocks being moved.Future developments of the Disk Driver Actor will solve this handicap by using the light weight remote procedure call mechanism provided by the CHORUS kernel.

— Second, the processing of asynchronous requests is not yet fully operational and thus has not been used. All requests are being processed synchronously, the performance gain from the deferred writes and the read-ahead has been lost.

Nevertheless, these figures prove the correctness of the CHORUS approach that consist in enabling servers to reside in supervisor space to achieve better performance.

6. Conclusion

We have been successful in splitting the disk driver from the CHORUS/MiX File Manager. These two servers have been encapsulated within UNIX processes. The disk driver can be loaded dynamically into the supervisor address space.

It has been possible to prototype all of this very quickly on top of CHORUS/MiX due to the pre-existing modularity and to the power of the tools provided by the CHORUS kernel.
In addition to the work in progress already mentioned, this prototype will enable us to implement the automatic loading of a driver upon the first open of a device, the File Manager being the parent process of the driver. The Disk Driver Actor will also be delivered to the Multi-Works project.

7. Acknowledgements

I would like to thank people that have contributed to the success of this experience: Roland Dirlewanger, Frédéric Herrmann, Denis Métral-Charvet, Didier Poirot, Marc Rozier and François Saint-Lu. I would also like to thank people that helped me with much worthy advices while writing this paper: Joëlle Madec, Allan Bricker, Michel Gien and Marc Guillemont.

8. References

[Rozier88a] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser, ‘‘CHORUS Distributed Operating Systems,’’ *Computing Systems Journal*, vol. 1, no. 4, The Usenix Association, (December 1988), pp. 305-370. Chorus systèmes Technical Report CS/TR-88-7

[Accett86a] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, ‘‘Mach: A New Kernel Foundation for UNIX Development,’’ *Summer Conference Proceedings 1986*, USENIX Association, (1986).

[Cherit88a] David Cheriton, ‘‘The V Distributed System,’’ *Communications of the ACM*, vol. 31, no. 3, (March 1988), pp. 314-333. Vsyst

[Mullena] S. J. Mullender, *The Amoeba Distributed Operating System: Selected papers 1984 – 1987*, CWI tract 41, Amsterdam, (1987). AMO87

[Armand89a] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier, ‘‘Revolution 89, or Distributing UNIX Brings it Back to its Original Virtues,’’ in *Proc. of Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, Ft. Lauderdale, FL, (5-6 October 1989), pp. 153-174. Chorus systèmes Technical Report CS/TR-89-36.1

[Armand90a] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier, ‘‘Multi-threaded Processes in Chorus/MIX,’’ in *Proc. of EUUG Spring’90 Conference*, Munich, Germany , (23-27 April 1990), pp. 1-13. Chorus systèmes Technical Report CS/TR-89-37.3

[Golub90a] Davic Golub, Randall Dean, Alessandro Forin, and Richard Rashid, ‘‘UNIX as an Application Program,’’ in *Proc. of Summer 1990 USENIX Conference*, USENIX, Anaheim, CA, (June 11-15, 1990), pp. 87-96. CS/EX-90-285 X90285

[Cherit90a] David R. Cheriton, Gregory R. Whitehead, and Edward W. Sznyter, ‘‘Binary Emulation of UNIX using the V Kernel,’’ in *Proc. of Summer 1990 USENIX Conference*, USENIX, Anaheim, CA, (June 11-15, 1990), pp. 73-86. CS/EX-90-284 X90284

[Cherit84a] D.R. Cheriton, ''A Uniform I/O Interface and Protocol for Distributed Systems,'' Resarch Report, Stanford U., (Dec 1984), p. 40. Z/2 SYS1201