Revolution 89
or
‘‘Distributing UNIX Brings it Back to its Original Virtues’’

François Armand, Michel Gien, Frédéric Herrmann, Marc Rozier

Chorus systèmes
6, avenue Gustave Eiffel, F–78182, Saint-Quentin-en-Yvelines (France)
Tel: +33 1 30 57 00 22, Fax: +33 1 30 57 00 66, E-mail: mg@chorus.fr

1. Introduction

The need to handle distributed computing in a general manner leads us to structure our operating systems functions in a much more modular way than it is done in today’s systems such as current UNIX kernels, and to provide facilities for dynamic reconfiguration so the system can be adapted to the variety of configurations needed in a distributed system. When applied to the UNIX kernel, such a "restructuring" leads to same kind of "revolution" that UNIX performed on the operating systems of the 70’s, i.e., to extract from the operating system all functions that can better be performed outside, and to leave in the kernel only those generic services that are necessary to provide higher level services, such as high level file access methods, command languages (shell) or system administration functions.

The CHORUS[1] architecture is designed to support new generations of open, distributed, scalable operating systems. It allows the integration of various families of operating systems, ranging from small real-time systems to general-purpose operating systems, in a single distributed environment.

The CHORUS architecture is based on a minimal real-time Nucleus that integrates distributed processing and communication at the lowest level. CHORUS operating systems are built as sets of independent system servers, that rely on the basic, generic services provided by the Nucleus i.e., thread scheduling, network transparent IPC, virtual memory management and real-time event handling.

The CHORUS Nucleus itself can be scaled to exploit a wide range of hardware configurations, such as embedded boards, multi-processor and multi-computer configurations, networked workstations and dedicated servers.

Operating systems (called Subsystems) implemented on top of this Nucleus currently include a UNIX[2] SYSTEM V[Herr88] and the "Emeraude"[Mino88] CASE/PCTE system. Work is currently in progress to implement Object-Oriented distributed Subsystems.[Alve88]

CHORUS–V3 is the current version of the CHORUS system developed by Chorus systèmes. Earlier versions were designed and implemented within the Chorus research project at INRIA between 1979 and 1986. Related work includes the V-system[Cher88] for the message-passing kernel, Mach[Rash87] and[Li86] for the distributed virtual memory, Topaz[McJo88] and Mach[Acce86] for the ‘‘threads’’, Amoeba[Mull87] for the global addressing, and the Bell Laboratories’9th Edition UNIX[Pres86, Wein86] for the uniform file naming.

1. CHORUS is a registered trademark of Chorus systèmes
2. UNIX is a registered trademark of AT&T

In: Proceedings of "Workshop on Experiences with Building Distributed (and Multiprocessor) Systems", 5-6 Oct. 1989, Ft. Lauderdale, FL, USA, pp.153-174.

CHORUS–V3 is written in C++ (and C). It currently supports 680X0 and 80386 based machines, with implementations on networked workstations and servers, as well as multi-processor configurations.

This paper outlines the architecture and implementation of UNIX kernel functions in terms of the CHORUS architecture concepts, based on the CHORUS Nucleus basic services. It focuses on the experiences drawn from it, the resulting benefits for users as well as for systems designers and maintainers, and the issues that still need to be considered.

The next section summarizes the CHORUS Nucleus’ basic abstractions and services as described extensively in. [Rozi88] Section 3 outlines the structure of a UNIX Subsystem, in terms of independent cooperating CHORUS servers, illustrating how one can make use of the Nucleus facilities in a UNIX context. Straightforward extensions in the services provided at the UNIX kernel interface level will also be presented. The remaining sections give examples of using the CHORUS architecture in typical system configurations and operating system experiments.

2. The CHORUS Architecture

2.1 Overall Organization

A CHORUS System is composed of a small-sized **Nucleus** and a number of **System Servers**. Those servers cooperate in the context of **Subsystems** (e.g., UNIX) to provide a coherent set of services and interfaces to their ‘‘users’’ (Figure 1).

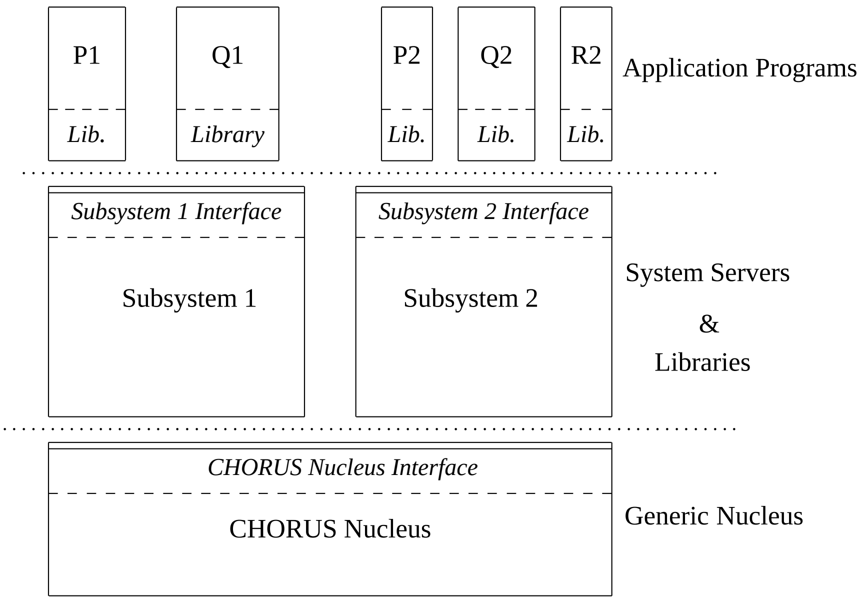

Figure 1. – The CHORUS Architecture

The CHORUS Nucleus (Figure 2) plays a double role:

1. Local services:
   It manages, at the lowest level, the local physical computing resources of a ‘‘computer’’, called a *site* by means of three clearly identified components:
   - allocation of local processor(s) is controlled by a *real-time multi-tasking executive*. This executive provides fine grain synchronization and priority-based preemptive scheduling,
   - local memory is managed by a *virtual memory manager*,
   - external events – interrupts, traps, exceptions – are dispatched by a *supervisor*.
2. Global services:
   An *IPC Manager* provides the communication service, delivering messages regardless of the location of their destination within a CHORUS distributed system. It may rely on external system servers

(i.e.,
Network Managers) to operate all kinds of network protocols.

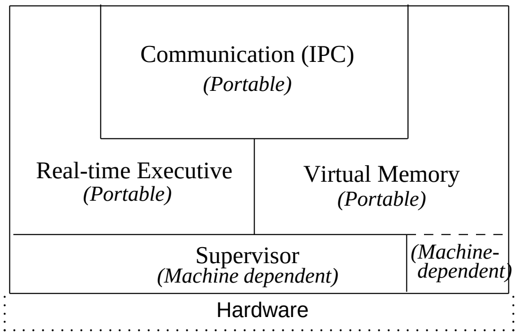

Figure 2. – The CHORUS Nucleus

2.2 The CHORUS Nucleus basic abstractions

The physical support for a CHORUS system is composed of a set of *sites* (‘‘computers’’, or ‘‘boards’’), interconnected by a communication *network* (i.e., a real network or a bus). A **site** is a tightly coupled grouping of physical resources: one or more processors, memory, and attached I/O devices. There is one CHORUS Nucleus per site.

The **actor** is the logical unit of distribution and of collection of resources in a CHORUS system. An actor defines a protected address space supporting the execution of one or more *threads* (lightweight processes) that share the address space of the actor. An address space is split into a *user address space* and a *system address space*. On a given site, each actor’s system address space is identical and its access is restricted to privileged levels of execution (Figure 3).

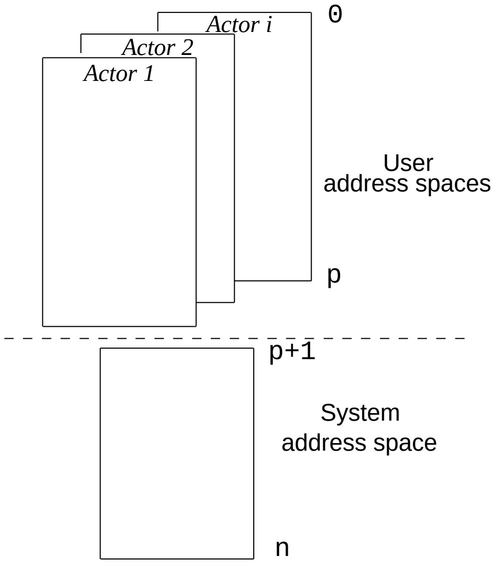

Figure 3. – Actor Address Spaces

Any given actor is tied to a site, and its threads are executed on that site. A given site may support many simultaneous actors. Since each has its own ‘‘user’’ address space, actors define protected *virtual machines*.

The **thread** is the unit of execution in a CHORUS system and is characterized by a *context* corresponding to the state of the processor (registers, program counter, stack pointer, privilege level, etc.). A thread is always tied to one and only one actor. These threads share the resources of that actor and no other actor. Threads are scheduled by the Nucleus as independent entities. In particular, threads of an actor may run in parallel on the many processors of a *multiprocessor* site. The scheduling of threads is preemptive, based on their fixed priorities.

Besides the shared memory provided by the actor address space, CHORUS offers message-based facilities (referred to as *IPC*) which allow any thread to communicate and synchronize with any other thread, on any site. The CHORUS IPC permits threads to exchange messages either *asynchronously* or by *demand/response*, also called *Remote Procedure Call* (*RPC*). Its main characteristic is its transparency with respect to the location of threads: the communication interface is uniform, regardless of whether it is between threads in a single actor, between threads in different actors on the same site, or between threads in different actors on different sites.

A **message** is composed of a (optional) *message body* and a (optional) *message annex*. Both are untyped string of bytes. Message passing is tightly coupled with the virtual memory mechanism to enable data transmission without copy.

Messages are not addressed directly to threads, but to intermediate entities called *ports* (Figure 4).

A **port** is an address to which messages can be sent, and a queue holding the messages received but not yet consumed by the threads. A port can only be attached to a single actor at a time, but can be attached to different actors successively, effectively *migrating* the port from one actor to another.

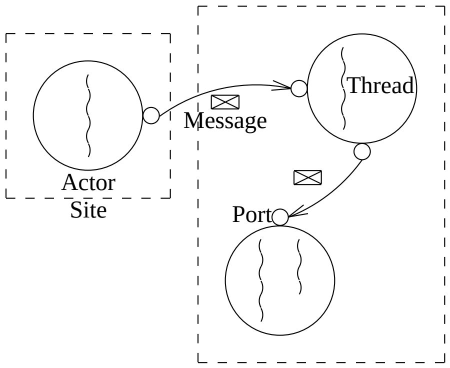

Figure 4. – CHORUS Nucleus basic abstractions

The notion of a port provides the basis for *dynamic reconfiguration*: this extra level of indirection between communicating threads, enables a given *service* to be supplied independently of a given actor. The servicing actor can be changed at any time, by changing the attachment of the port from the actor holding the initial thread to the actor holding the new one.

A **group** of ports connects those ports to a *multicast* facility: it allows one thread to communicate directly with an *entire group of threads* (via a group of ports); it provides also *''functional''* access to a service by selecting a server from a group of (equivalent) servers. A group is built by dynamically inserting ports into, and removing them from, the group.

Ports are globally designated with *Unique Identifiers* (UI's). A **UI** is unique in a CHORUS system. The CHORUS Nucleus implements a localization service, allowing threads to use these names without any knowledge of the location of the designated entities. UI's may be freely exchanged between actors.

Global names for other types of objects are based on UI's, but hold more information, such as protection information. These names are called *capabilities*. [Tane86] A **capability** is made of a UI and an additional structure, the *key*. When objects are Nucleus objects (e.g., actors), the UI is the global name for the object, and the key is only a protection key. When an object is managed by an external server (e.g., memory segments), the UI is the global name of a port of that server, and the semantics of the key are defined by the server. Generally, the key identifies the object within the server and holds the protection

information.
As with UI’s, capabilities may be freely exchanged between actors.

2.3 Virtual Memory Management

The CHORUS memory management service[Abro89, Abro89a] provides separate address spaces (if the hardware gives adequate support), associated to actors, called **contexts**, and efficient and versatile mechanisms for data transfer between contexts, and between secondary storage and a context. The mechanisms are adapted to various needs, such as IPC, file read/write or mapping, memory sharing between contexts, and context duplication.

CHORUS memory management considers the data of a context to be a set of non-overlapping **regions**, which form the valid portions of the context.

Regions are mapped (generally) to secondary storage objects, called **segments**. Segments are managed outside of the Nucleus, by external servers called segment **mappers**. These manage the implementation of the segments, as well as the protection and naming of segments.

2.4 The Supervisor

The CHORUS Nucleus offers the following basic services to allow system actors to handle hardware events such as interrupts, traps and exceptions:

System threads may connect handlers (e.g., routines in the address space of their actor) to hardware interrupts. When an interrupt occurs, these handlers are executed. Several handlers may be simultaneously connected to the same interrupt, with control mechanisms to order or stop their invocation. After acknowledging the interrupt, handlers can communicate with other threads using asynchronous IPC or synchronization primitives provided by the Nucleus.

System actors may also connect routines to trap invocations, either as one routine or as an array of routines. In the latter case, the handler actually invoked is specified by a "service" number stored in a register of the machine.

Finally, an exception port or an exception routine can be associated with an actor, thus permitting Subsystem actors to deal with faults occurring within other actors.

TABLE 1. − Supervisor Interface

Supervisor interface	
`svConnect`	*Connect an interrupt or trap handler*
`svDisConnect`	*Disconnect an interrupt or trap handler*
`svCallConnect`	*Connect a trap handling table*
`svCallDisConnect`	*Disconnect a trap handling table*

3. The UNIX Sub-System

3.1 Overall structure

UNIX facilities may logically be partitioned into several classes of services according to the different types of resources managed: processes, files, devices, pipes, sockets. The design of the structure of the UNIX Subsystem in CHORUS puts emphasis on a clean definition of the interactions between these different classes of services in order to provide a true modular structure.

The UNIX Subsystem has been implemented as a set of System Servers, running on top of the CHORUS Nucleus. Each type of system resource (process, file, etc.) is isolated and managed by a dedicated system server. Interactions between these servers are based on the CHORUS IPC which enforces clean interface definitions (Figure 5).

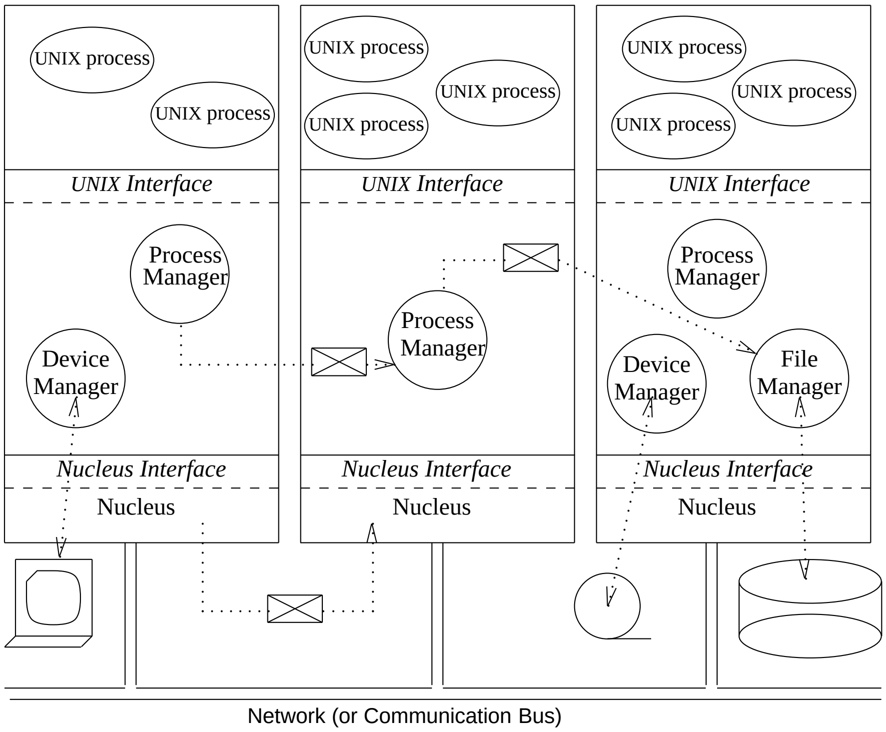

Figure 5. – UNIX as a Set of Independent Servers

Several types of servers may be distinguished within a typical UNIX Subsystem:

– **The Process Manager (PM)**
The Process Manager (PM) executes services directly related to UNIX process management (creation and destruction of processes, signals, etc.). The UNIX services have been extended to provide transparent access to distributed resources, so the PM’s on the different sites of a network cooperate to provide remote services (such as remote `kill` or remote `exec`).
The PM manages the system context of each process. When the PM is not able to serve a UNIX system call by itself, it calls other servers as appropriate.

– **The File Manager (FM)**
The File Manager (FM) performs file management services. The current version is compatible with SYSTEM V.2 services and physical disk layout. New versions, compatible with SYSTEM V.3.2 and BSD 4.3 respectively are currently being integrated into CHORUS.
The FM also acts as a CHORUS *external mapper* for distributed virtual memory management by performing the `page_in`/`page_out` requests issued by the CHORUS Nucleus Virtual Memory Manager.

– **The Device Managers (DM)**
The Device Managers (DM) manage asynchronous lines, bit-map displays, *pseudo-ttys*, etc. and implement the UNIX *line disciplines*. Several DMs can run simultaneously on one site servicing different peripheral devices.

– **The Pipe Manager (PIM)**
A Pipe Manager implements UNIX pipe management and synchronization. `Open` requests for named pipes, received by File Managers are forwarded to a Pipe Manager. Pipe Managers may be active on every site, thus reducing network traffic when pipes are invoked on diskless stations.

– **The Socket Manager (SM)**
The Socket Manager implements BSD 4.3 socket services, providing access to TCP/IP protocols.

Those system servers can run either in *User space* or in *System space*. Those needing to connect some of their routines to *traps* (like the PM) or to execute privileged instructions (like I/O operations) run in system space. Loading a server in system space also has some impact on the performance of the server as it avoids extra memory context switches when the server is invoked.

3.2 Functional extensions

The interface offered by the UNIX Subsystem on a given machine, can be made binary compatible (i.e., at the executable code level) with a standard UNIX system taken as a reference (currently System V Release 3.2 on AT/386), to ensure complete user software portability. In addition, UNIX drivers can be integrated into a CHORUS Server with minimum effort.

CHORUS also provides extensions to the UNIX interface to take benefit of the distributed nature of the system and of the underlying CHORUS Nucleus services.

3.2.1 File System extensions

The naming facilities provided by the UNIX file system have been extended, to permit the designation of services accessed via Ports.

Symbolic Port Names (new UNIX file type) can be created in the UNIX file tree (Table 2). They associate a file name to a port Unique Identifier (this is very similar to UNIX device designation). When such a name is found during the analysis of a pathname, the corresponding request is forwarded to the port – marked with the current status of the analysis.

TABLE 2. – UNIX Symbolic Port System Calls

Symbolic Port System Calls	
`symport`	*create a symbolic port*
`readport`	*get the Unique Identifier associated with the symbolic port*
`lstat`	*do* `stat(2)` *on the symbolic port itself*
`unlink`	*unlink the symbolic port itself*

User written servers as well as system servers can be designated by such symbolic port names, thus allowing "users" to make extensions to the system dynamically (see Section 4.5). In particular, this is used to interconnect file systems and provide a global name space. For example, in Figure 6, ‘‘pipo’’ and ‘‘piano’’ are symbolic port names.

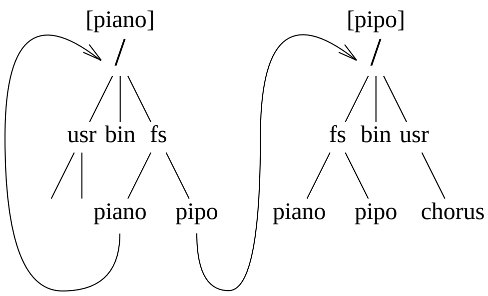

Figure 6. – Interconnection of File Trees

3.2.2 Process Management extensions

These extensions have been introduced to traditional UNIX services:

- The basic extension to process management is to enable remote creation (`fork(2)`) or execution (`exec(2)`) of processes. A "creation site" information has been added to the system context of UNIX processes. This information is inherited through `fork(2)` and `exec(2)`. It may be set by means of a new system call: `csite (SiteId)`. This information is used when a process is forking or exec'ing: on a `fork(2)`, the child process will be created on the site specified by `SiteId`. On `exec(2)`, the process will start the execution of the new program on the site specified by `SiteId`.

Creating or moving processes on any site, implies that process identifiers are unique over the distributed system. Process identifiers used by UNIX servers are 32 bits long. UNIX processes can manipulate either PIDs of 16 bits (for binary compatibility reasons) or 32 bits which will allow them to address signals to remote processes. The new `pcntl (LONGPID)` system call sets a system context flag which enables a process to manipulate PIDs 32 bits long.

- At `exec(2)` time, processes can be dynamically loaded into the system space, provided that the text and data regions that they need are free. Such a process will execute at a privileged level, thus being able to execute I/O instructions.
- Processes can lower or raise their priority, thus allowing real-time applications to run on the UNIX Subsystem (see 4.4).

3.2.3 Other extensions

It is natural to provide UNIX processes with access to some of the services offered by the CHORUS Nucleus i.e., IPC, Virtual Memory and Threads. Such access is not provided by directly invoking the Nucleus but rather through the UNIX Process Manager, in order to eliminate inconsistencies. For example, if a UNIX process could create a thread by directly invoking the CHORUS Nucleus without the Process Manager knowing about it, this thread would not be able to issue UNIX system calls correctly.

Therefore, some CHORUS Nucleus services are not available at the UNIX Subsystem interface (e.g., no actor creation or deletion primitives), and some restrictions and controls are performed: e.g., forbid the creation of threads inside other UNIX processes and the use of the UNIX process identifier instead of the CHORUS actor Unique Identifier in calls such as `portMigrate`.

To clearly distinguish between the two levels of interfaces, UNIX primitives allowing access to CHORUS services have been prefixed by "`u_`" (e.g., `u_portCreate` instead of `portCreate`).

3.2.3.1 Virtual Memory services

UNIX processes can use the Virtual Memory services of the CHORUS Nucleus to create regions, map segments within a region, share regions, etc. They can thus gain access to the physical memory (e.g., for mapping bitmap memory).

3.2.3.2 Inter Process Communication

UNIX processes can create ports, insert ports into groups, and send and receive messages. They can migrate ports from one process to another. CHORUS IPC mechanisms allow them to communicate transparently over the network. Applications can therefore be tested on a single machine, and then distributed throughout the network, without any modification necessary to adapt to a new configuration. Using port migration or group facilities provides a sound basis for doing dynamic reconfiguration and developing fault-tolerant applications.

3.2.3.3 Multi-threaded UNIX Processes

Multiprogramming within a UNIX process is possible with **u_threads**. A u_thread can be considered as a lightweight process within a standard UNIX process. It shares all the process resources and in particular its virtual address space and open files. Each u-thread represents a different locus of control.

When a process is created by `fork(2)`, it starts running with a unique u_thread; the same situation occurs after `exec(2)`; when a process terminates by `exit(2)`, all u_threads of that process terminate with it.

A set of signal handlers is associated with each u_thread. Signal sent on an exception are delivered to the faulty u_thread (only); alarm signals are delivered to the u_thread which set the alarm; all other signals are broadcasted to all u_threads of the process. Signal handlers are executed on the stack of the u_thread

which
set the signal handler. Thus full consistency with existing signal handlers is maintained. U_threads may issue UNIX system calls. For reasons of simplicity (i.e., efficiently insuring consistency of the process system context), these are serialized except blocking system calls such as `read(2)`, `write(2)`, `pause(2)`, `wait(2)`, `u_ipcReceive(2)` and `u_ipcCall(2)` (i.e., those interruptible by signals).

3.3 Implementation

3.3.1 Structure of a UNIX Process

A UNIX process can be viewed as one thread of control executing within one address space. Therefore each UNIX process is implemented as one CHORUS actor. Its UNIX system context is managed by the Process Manager. The actor address space is divided into memory regions for text, data and execution stacks.

In addition, the Process Manager attaches one *control port* and one *control thread* to each actor implementing a UNIX process. The control port and the control thread are not visible to the user of that process.

Control threads executing within process contexts share the process address space and can easily access and modify the core image of the process (e.g., stack manipulations on the reception of a signal, text and data access during debugging). They are also ready to handle asynchronous events received by the process (mainly signals). These events are implemented as CHORUS messages received on the control port (Figure 7).

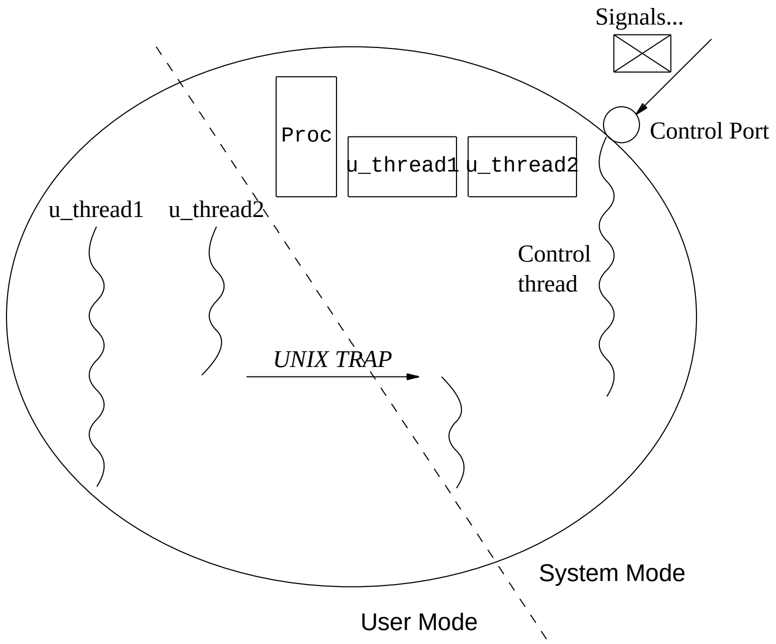

Figure 7. – UNIX Process as a CHORUS Actor

Because a process can be multi-threaded, the UNIX system context attached to one process has been split into two system contexts: one process context (`Proc`) and one u_thread context (`u_thread`).

Most services implemented outside of Process Managers are file related services. However, the file context of a Process (e.g., current and root directories, open files, `umask` and `ulimit` informations) are kept in the `Proc` structure, held by the Process Manager. This implies a specific protocol between PM’s and other servers, as shortly outlined in Section 3.3.4.

Both system contexts `Proc` and `u_thread` are maintained by the Process Manager of the current process execution site. These contexts are accessed neither by the CHORUS Nucleus nor by other system servers. On the other hand the UNIX Subsystem is unable to see the internal Nucleus structures associated with actors and threads, the only way to access them is through Nucleus system calls (this is

essential
for allowing multiple Subsystems to co-reside on top of the same CHORUS Nucleus).

3.3.2 Process environment known by its set of ports

The semantics associated with ports by the CHORUS Nucleus – unique and global naming, addressing by IPC with location transparency – makes them extremely useful for designating system entities. The main advantages of using ports are the indirection that they provide between the process and its environment, and the robustness against the evolution of configurations. Port names stored in the process context are always valid whether the process itself migrates to another site (i.e., `exec` to a remote site) or if some of the entities to which they are related to migrate.

Used directly or embedded within capabilities, ports constitute the main part of a process environment. Embedded in capabilities, ports are used to designate process resources: e.g., open files or segments mapped into the process address space (`text`, `data`). But ports are also used directly to address processes.

Resources and capabilities

Every resource (managed by a Server) used by a process is designated internally by a capability: open file, open pipe, open device, current and root directories, text and data segments, etc. Such capabilities may be used to create regions in virtual memory; thus their structure is the one exported by the CHORUS Nucleus.

For example, opening a file associates the capability sent back by the appropriate server to the correct file descriptor. The capability is built with the port of the server that manages that file and the reference of the open file within the server. All requests on that open file (e.g., `seek(2)`, `close(2)`) are translated directly into a message and sent directly to the appropriate server.

Because the server of a resource is designated by a port, and because the localization of a port is transparent as part of the CHORUS IPC, the UNIX Subsystem does not have to locate UNIX servers.

Capabilities are computed and sent back by the servers. A server can thus delegate a service to another server, without clients knowing which actual server serves its requests.

3.3.3 The Process Manager

All of the UNIX Subsystem code concerning process management, signal handling, and the interface for the system calls accessible from a process, are in a single actor: the Process Manager (PM).

This actor is loaded when booting the system. Its code and data areas are initialized in the system memory space. The presence of a PM inside the system area makes it possible to implement system calls by using `traps`, as in UNIX (u_threads are running PM code after each system call), thus allowing binary compatibility with other UNIX systems.

The PM actor has the following resources:

– A *port* for receiving RPC requests addressed to the PM (remote `kill` and `exec`). This port is inserted in the *static group* of PM ports, used for locating a process.

– A *thread* dedicated to processing requests received on that port.

– A *thread* used for managing alarms. That thread is woken up each time an alarm arises and it sends a message to the control port of the actor that owns the u_thread which set the alarm.

– The data area of the actor. It includes, in particular, the Proc and the u_thread structures.

– A scratch area used to send and receive messages or to access stack areas of user processes (e.g., for mapping/demapping operations).

– The code area of the actor. Located in the system memory space, it is shared by all processes and executed by PM threads when processing remote requests and handling alarms, by u_threads when doing a system call, or by control threads (one per process) upon receipt of asynchronous messages addressed to processes (e.g., signals, children, death).

3.3.4 The File Manager

The FM is a system actor that has two ports on which it receives request messages. One of these ports deals exclusively with messages for paging virtual memory. The other port receives all other requests: UNIX services, cooperation messages between PM and the FM, between Device Managers (DM) and the FM, and between the FM and other FMs in a distributed system. This port is called the "UNIX port"of the FM.

Once the initialization phase is over, several threads execute inside the FM. These threads process messages from one or the other of the two ports (Figure 8).

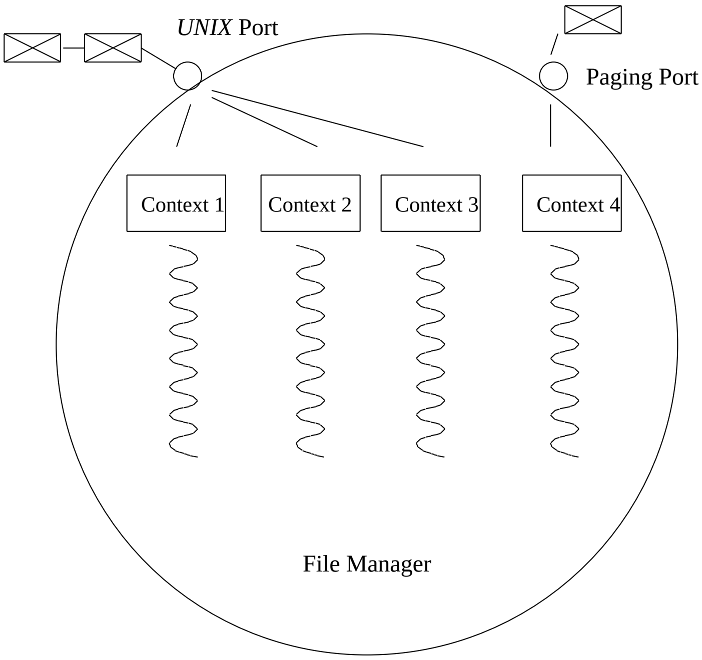

Figure 8. – File Manager Dynamic Structure

The general execution scheme for these threads is:

1. wait for a message to process,
2. initialize the thread context from the contents of the message,
3. invoke the required service,
4. prepare the reply message, and send it to the original requester,
5. return to (1).

The FM contains state information similar to that of a traditional UNIX file system. It owns and manages the following structures:

- A table of open files (one entry per `open(2)` performed), that contains in particular the flags used when opening the file (read and/or write access, etc.) and the current position in the file.
- A table of *inodes*, containing the memory images of the disk descriptors for the files currently in use.
- A table of the mounted volumes containing the volume descriptors of the disks currently mounted.
- A cache of the disk blocks, allowing the FM to minimize the number of physical disk accesses.

In addition, each thread that processes requests has an associated process context structure that simulates the system context that would be present in a traditional UNIX kernel (i.e., the U area). This context contains, for example, the identification of the user on whose behalf the thread is performing the request, the parameters of the request, and the global scratch variables equivalent to those of a UNIX kernel file system.

Because the context of the process on whose behalf the FM is processing the request, is not directly accessible to the FM, context information needed to serve the requests are included into the request messages, together with the system call parameters. The server includes in the replies those information

© Chorus systèmes, 1990

necessary
to update the file context of the process. Such a scheme is illustrated in Figure 8. Other servers such as Pipe, Device, or Socket Managers operate under a similar scheme.

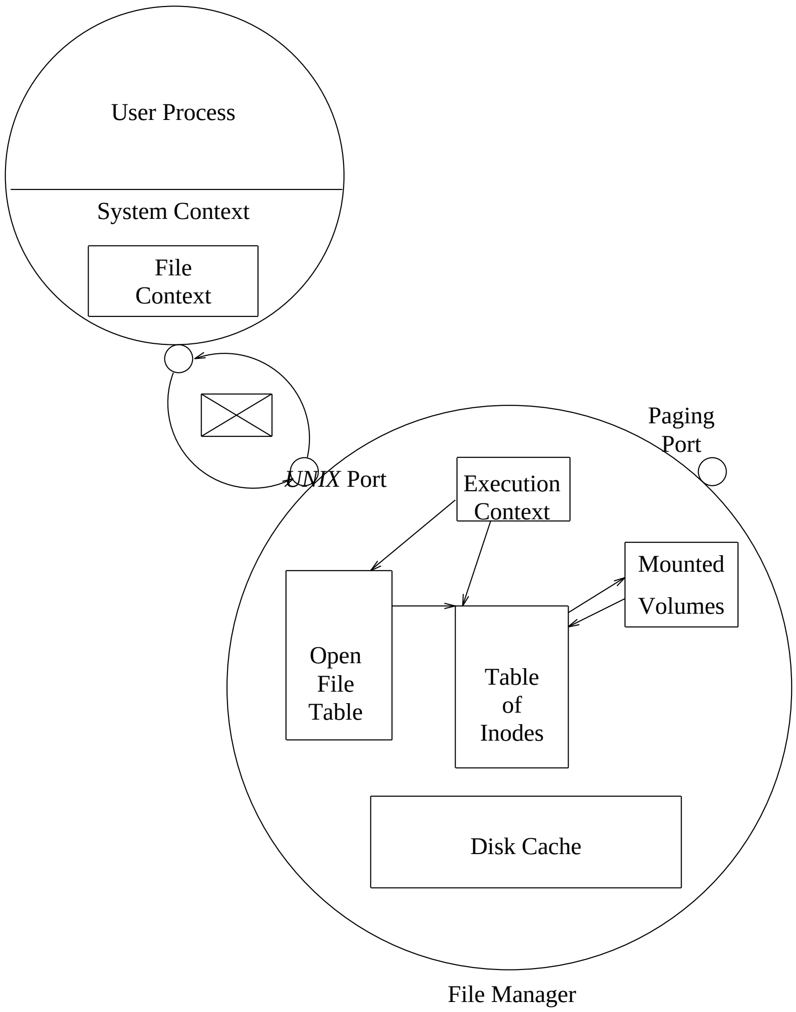

Figure 9. – Process and File Manager File Context

4. UNIX brought back to its original virtues

4.1 A tool-kit system

The structure of the UNIX Subsystem of CHORUS brings back to UNIX some of its original characteristics which have been gradually worn away by the thousands of hacker×years spent introducing new features into a monolithic kernel. The same ideas that UNIX had been promoting regarding the development of software tools have been applied by CHORUS to the operating system itself. They can be summarized as:

- make system servers implement only one type of service very simply and efficiently, rather than a lot of complicated features inefficiently,
- adapt existing servers rather than redoing everything from scratch, and fill the gaps by developing only those servers which are missing, when you want to build a new operating system (or extend an existing one).

Some of the servers in the UNIX Subsystem have been written from scratch (e.g., Process Manager, Socket Manager), while others have been adapted from existing UNIX kernel code (e.g., File Manager, Device Manager). In both cases the interdependencies and functions of each of the servers have been carefully designed, so that they can be combined in various ways to adapt the behavior of the resulting system to its user’s needs. The servers have been made as flexible as possible so that they can be

dynamically
configured.

Some of the areas which most benefit from such a design and which will be examined in the following sections are:
- the static configuration of a distributed system,
- the dynamic (re)configuration of a distributed system,
- the ability to change the system behavior,
- the configuration of the system interface and semantics.

Most of these capabilities come from the basic services provided by the CHORUS Nucleus, but also from the way these services are used by the upper layers of the system.

4.2 Modularity, static configuration and distribution

As the UNIX subsystem is composed of a collection of servers, it is straightforward to adapt it to the hardware configuration of the system, or to the needs of the applications which run on such a configuration:
‘‘No disk on your machine?’’... ‘‘Don’t take the File Manager’’...
‘‘No Terminal connected to your embedded system?’’... ‘‘Don’t load the Device Manager’’...
‘‘Your application uses sockets but no pipes?’’... ‘‘Take the Socket Manager, but not the Pipe Manager’’...
This is possible because these servers are truly independent from one another and because their only interface to their clients is through the CHORUS IPC.

4.2.1 Typical configurations

4.2.1.1 Standalone Machines

On a standalone machine, obviously, there is no need of network protocols, so the Network Manager need not to be part of the system.

4.2.1.2 Diskless Workstations

On a diskless workstation, there is no need of a File Manager. Only a Nucleus, a Network Manager, a Process Manager, a Device Manager (to support a bitmap and pseudo-ttys) and possibly a Socket Manager are required to provide a full UNIX environment. UNIX file system calls are converted into IPC requests by the PM, thus allowing transparent access to File Managers running on remote disk servers.

The Pipe Manager resides on the diskless workstation, but this is not mandatory. If it is not there, another equivalent server in the distributed system will serve the pipe requests of that station. Loading it on the station itself only provides better response time for accessing pipes, because it avoids accessing the network.

4.2.1.3 Multi-computers

From the point of view of a distributed system like CHORUS, the structure of a multi-computer (e.g., a hypercube) is very similar to the structure of a network of servers and workstations. The same configuration choices can be made: loading drivers only on the nodes where they are useful, loading a Socket Manager on the nodes providing connections with the outside world, loading a File Manager where disks are located. Only a Nucleus, a Network Manager and a Process Manager need to be present on each node to make that node look like a full UNIX system to application programs (on nodes running only one process of the application, this can actually be reduced to a simpler run-time system).

In fact, the version of the Network Manager running on a node not connected to an external network need not implement all the network protocols but only those handling inter-node communication. Network Managers running on nodes that provide access to an external network, must provide both families of protocols. This system architecture is being used on the EuroWorkStation developed in the EWS Esprit II Project 2569.

4.2.1.4 Multi-processors

The CHORUS kernel can run on symmetric multi-processor machines, providing its actors with uniform and simultaneous access to the processors.

The modularity of the UNIX subsystem allows benefiting directly from this facility: because UNIX services are implemented as independent servers, no synchronization is required between these servers. Thus, on a four processor machine, each processor could have a thread running from a different UNIX server (e.g., PM, FM, DM, SM). This provides a very simple multiprocessing of UNIX services without having to worry about adding synchronization schemes to those UNIX servers which have been directly derived from current UNIX kernel code (FM, DM).

Moreover, some of the new servers, such as the Process Manager, have already been structured so that they can be themselves multiprocessed. Thus, several processes can invoke simultaneously a UNIX system call. On a four processor machine, two processes can `fork(2)`, another can issue an `open(2)`, and still another one a `read(2)`, all in true parallelism. Synchronization on global tables of the Process Manager is done at a fine level of granularity.

Regarding multiprocessing, servers implemented from UNIX code have the same level of granularity as in current UNIX kernels. Finer levels of granularity will be introduced in further releases of the system, using the synchronization primitives offered by the CHORUS Nucleus.

4.2.1.5 Embedded systems

For real time applications running in embedded systems, there may be no need or use for services other than those offered by the CHORUS Nucleus, the Process Manager and the Socket Manager. These will provide such applications with access to process and thread primitives, IPC, memory management, and connection to the hardware. Communication with the outside world can be done through the Socket Manager, using the services of the Network Manager.

These embedded applications may run in an environment without any other machine managed by CHORUS. In order to offer more flexibility (file access) and dynamism (loading/unloading programs, remote debugging) to such embedded applications, file services (including pseudo tty’s), can be provided through a very simple File Manager which maps all CHORUS IPC file requests to socket communication. The requests carried through a socket connection are then processed by a UNIX process acting as remote server and running on any UNIX system.

4.2.2 Examples

The adaptability of the UNIX subsystem is clearly illustrated by the three following real cases.

4.2.2.1 Fault tolerant documents database server

A document database server that runs on a Motorola 68030 board plugged into a MacII[3] running CHORUS has been developed by an independent company. The database application runs on the board which also supports the disks (mirrored disks and/or an optical jukebox). The only services needed by the application are disk management and file access. Access to the database services is provided to the outside world (i.e., the MacII and other clients connected to the MacII by a network) via common memory shared by the MacII and the 68030 board.

The underlying system is composed of a CHORUS Nucleus (without the Network Manager) and a File Manager. A library was developed (from the Process Manager code, in one month) to transform every UNIX file system call into a CHORUS IPC request. The UNIX-like file context is thus managed in user’s space in the same way as I/O streams are managed in the standard C library. Avoiding a full Process Manager saves memory space and provides better response time on file access by avoiding traps.

4.2.2.2 X terminal

CHORUS is being used in an X terminal product built by an independent company. The only program that runs in such a configuration is the X–server, which serves X–window requests coming from clients on other machines. It is to run in environments without other CHORUS machines.

3. MacII is a trademark of Apple Computers

The X terminal system is composed of a CHORUS Nucleus, a Network Manager, a Process Manager, a Socket Manager and a Device Manager (for keyboard and mouse drivers). There is no need for a Pipe or File Manager. Because the X–server still does some `open(2)` calls at init, to access devices, a small dedicated File Manager has been developed (in less than a week) to serve these specific requests.

4.2.2.3 EuroWorkStation

The EuroWorkStation being developed in the EWS project previously mentioned is a high level scientific workstation. It is organized around a SPARC based symmetric multi-processor (as the main board(s)) which uses a Multibus II. An Intel 80386 based coprocessor board can be added on the MultiBus II. Access to Ethernet, FDDI and SCSI bus are also provided. Bitmap display is done (remotely) to an X–terminal. Planned coprocessors include 3D graphics, Lisp and a simulation engine.

The CHORUS system will run on such a configuration with a full UNIX Subsystem on the main board(s). On the co-processors, only a CHORUS Nucleus, a Network Manager (adapted to the Multibus II) and a UNIX Process Manager will be loaded. This will allow users to dynamically load programs on the co-processor from the basic workstation. To access the specific hardware of the coprocessors, an adapted version of the Device Manager will be loaded on each co-processor.

4.3 Dynamic Configuration

4.3.1 Sub-system configuration

The only UNIX server that needs to be loaded at boot time is the Process Manager, (and the File Manager if there are disks connected to the machine). With the CHORUS Nucleus it provides enough services to dynamically load other servers when needed. A very simple access to the system console (if any) is provided by the CHORUS Nucleus. Of course, UNIX-like tty line disciplines are not implemented in that case. This allows a shell to run on the console or to provide input/output on a terminal for processes which do not need UNIX terminal management.

The dynamic loading of UNIX servers is achieved through the standard UNIX interface: `fork(2)` and `exec(2)`. The Device Manager, the Pipe or the Socket Managers can be loaded by `init(1)` from the UNIX System V "*/etc/inittab*" file. This results from two services offered by the Process Manager to the super-user:

– the ability to load a UNIX process into the system address space, as specified at link time. Of course, if the virtual addresses needed by the process to be loaded are already used, `exec(2)` will fail.

– the ability to dynamically connect user specified routines to hardware interrupts. Such routines are invoked each time the interrupt occurs, until the routine has been disconnected from that interrupt.

Each device driver may be implemented in a separate Device Manager (e.g., for bitmaps, RS232 interfaces, tapes), these drivers can be loaded only when they are needed and can then be unloaded when they become useless. These Device Managers and the Pipe and the Socket Managers as well, are in fact UNIX processes and thus can take advantage of the UNIX services offered by the Process Manager and the File Manager, allowing them, for example, to record events into log files using standard UNIX system calls.

This dynamism can also be used to change the UNIX behavior of the subsystem. This was used in the Aphrodite Esprit Project 1535[Mino88] to build a host/target development, remote execution and debugging environment for real-time applications. A simple window manager was developed on top of the UNIX subsystem. For reasons of efficiency, this Window Manager (loaded as a UNIX process) catches interrupts directly from the mouse and the keyboard of an AT/386 computer. When the Window Manager is running, it diverts every interrupt from the Device Manager; the UNIX shell is thus blocked waiting for input. When the Window Manager becomes useless, quitting it disconnects the interrupt from the Window Manager routine and the Device Manager continues to work unaffected.

4.3.2 Server configuration

Since UNIX services are implemented by servers built on top of the CHORUS Nucleus interface, it is very easy to dynamically adapt the resources of each server to the actual needs of the application. This dynamic configuration relies on the following:

4.3.2.1 Adding threads in servers

When a process is created (or servers loaded at boot time), it runs only one thread. Other threads may be created dynamically. For example, the File Manager and the Device Manager create additional threads during their initialization phase. As servers are accessed through IPC or traps, their respective number of threads can be adapted to their varying needs without stopping the system (to reload newly configured servers). Of course, servers may also delete useless threads.

In particular, such a scheme is used by the File Manager when a diskless site is coming up. The File Manager is told to create new threads in order to serve such a site. When the site shuts down, those threads become useless and are deleted.

Currently, these configuration parameters are monitored by the system administrator using dedicated commands. Another approach would be to let servers create a new pool of threads themselves when the number of idle threads goes under a low water mark and to delete them when this number raises over a high water mark. Such configuration issues are discussed in Section 5.

4.3.2.2 Space management in servers

Adding threads in UNIX servers is not the only configuration issue. One must also configure the memory resources to the size appropriate to the use of the system. For example, if one wants to raise the maximum number of processes that may run simultaneously on a processor, the Process Manager must resize its Proc table. Dynamically resizing tables requires allocating memory space for new entries, and algorithms for allocating, freeing and searching entries that do not depend upon the physical organization of the tables. Space allocation within an actor can be done by invoking `rgnAllocate` with the size needed. This call returns the address of the newly allocated memory region.

The second problem is solved (or eased) by the use of C++. Basic tools for managing "*pools*" of elements have been developed. The process table is such a pool. When creating a new process, the Process Manager invokes "`PROC.allocate`" to get a free entry. When the process exits, it calls "`PROC.free`" to free the entry in the table. The implementation of the pool is hidden by these functions. The current implementation relies on linked list mechanisms. This pool mechanism is used by the servers for every new table that has been introduced. It is not used for tables allocated and managed by C code coming from an existing UNIX kernel implementation. This will be done in a future release of the system.

Having servers implemented as actors and allocating their internal tables in virtual regions eases the management of the usage of physical memory. Servers such as the Socket Manager, the Pipe Manager and even the Process Manager may be paged out without disturbing the service. Servers that connect code to interrupt are locked in memory to avoid page faults upon reception of an interrupt.

4.4 Real-Time operation

Making use of the real-time scheduling provided by the CHORUS Nucleus facilitates development (static or dynamic) of different scheduling policies in the Subsystem. The UNIX Subsystem of CHORUS takes advantage of this facility to provide a real-time execution environment.

4.4.1 Changing the priorities of a server

The CHORUS Nucleus schedules threads on a fixed priority basis. Priorities range from 1 (the highest) to 255 (the lowest). Threads running with a priority between 128 and 255 are time-sliced at the same priority level. In a usual CHORUS configuration, UNIX processes run at priority 128 and UNIX servers at priorities 64 to 68. As explained earlier, this gives the user the ability to run real-time processes with a higher priority than standard UNIX processes; they may even run with a priority higher than the UNIX servers!

If the range of priorities used by the UNIX servers is not adequate for a given system, the priorities of the UNIX servers can be changed either statically by recompiling the servers, or dynamically by sending them a request to change their priority through the `threadPriority` system call. In both cases, priorities of these servers may be lowered or raised as needed. In any case, attention should be given to the consistency of the new set of priorities used by the servers: it may not be very meaningful to have UNIX servers running at the lowest priority, while standard UNIX programs are running at the highest one...

4.4.2 Delaying interrupt processing

UNIX servers which deal with hardware interrupts (File Manager, Device Manager) execute interrupt code at interrupt level, just as in a standard UNIX kernel, providing equivalent response times to those provided by such UNIX kernels.

However, the File Manager and the Device Manager may also run in another mode. Upon reception of an interrupt they can just post an event to a dedicated thread (named the "*Interrupt Thread*") created at init time, and then return, after the interrupt level has been acknowledged. Posting an event can be done by means of synchronization primitives offered by the CHORUS Nucleus. The real interrupt routine of the driver will be executed when the Interrupt Thread gets the processor, depending on its priority. This "delayed processing of interrupts" minimizes the time during which interrupts are masked, leaving more time for real-time processes to run and deal with their own interrupts. In this mode, critical sections management inside the File Manager (or Device Manager) is done without actually masking the interrupts. Of course, this mechanism requires additional scheduling for each interrupt which processing has been delayed. This implies that the UNIX response time will be affected, therefore this is only useful when executing real-time applications.

The system administrator can dynamically change interrupt processing from immediate processing (as in standard UNIX implementations) to delayed processing mode, and vice versa. In addition, the priority of the thread managing the interrupts can be fixed dynamically. This feature has been implemented in the File Manager as well as in the Device Manager. The processing mode of interrupts can also be chosen or even be frozen when compiling the server.

This functionality allows the development of real-time applications in a standard UNIX environment while editing or compiling the application. Once the application has been written, prior to executing or testing it, the priorities of the UNIX servers can be lowered, thus minimizing masking periods to enable the application to react correctly. When the application is finished running, the priorities of the UNIX servers can be reset to their previous values making interrupts immediately processed and recovering the initial system behavior. Such facilities avoid the necessity of stopping and reloading the system(s) (the UNIX development system and the real-time execution system). This mechanism is roughly analogous to that provided by the UNIX shell with job control, which allows one to stop processes and then to restart them later. Here the system is not stopped but only "niced".

4.5 Extending system services

Another aspect of the flexibility provided by the CHORUS implementation of UNIX is the ability to dynamically tailor the services offered by the system to the user's needs.

4.5.1 Adding system calls

As illustrated earlier, the UNIX Subsystem as been extended in two ways: by a minimum set of extensions to standard UNIX interfaces for distributed environments, and by CHORUS extensions to provide UNIX processes with the IPC, threads and virtual memory services offered by the CHORUS Nucleus. In fact, the second category of extensions may be made accessible or not to UNIX processes.

The Process Manager uses `svCallConnect` to connect routines to traps (i.e., system calls). But UNIX services and CHORUS specific services are not implemented through the same "*sysent*" table, in order to facilitate the adaptation of the Process Manager to provide binary compatibility with a given UNIX implementation on a given hardware.

In addition to the standard UNIX interface, the Process Manager provides a service which permits extensions to be made available or unavailable, thus tailoring the interface to particular needs. This capability will be completed with dynamic loading of pieces of the Process Manager code, when compilers generating position independent code are more widely available.

This connection of routines to traps can also be used by UNIX processes loaded in system space to extend the current interface with new services accessible through traps. This allows extensions to the UNIX system to be written or provides a new system interface (e.g., an Object Oriented System) on top of the CHORUS Nucleus simultaneously with the UNIX interface.

4.5.2 Enriching the UNIX semantics

Another way to provide more services is to use the file system’s symbolic port mechanism. This has been used in a research project[Coyo89] that transparently provides duplicated files to UNIX. It does so without modifying either the interface, existing programs, or even the Process Manager or the File Manager.

At init time, the Duplicating Server creates a port and records it in the UNIX file system. Each time the File Manager encounters this port when analyzing a pathname in one of its requests, it redirects that request to that port. The Duplicating Server can then examine the request, duplicate it and send two resulting requests to two File Managers . After receiving the two responses, it replies to the client process, as if it was the initial File Manager.

Example:

If the sub-tree of files to be duplicated starts at the directory "/users/fa/srcdir", the server creates the symbolic port "/users/fa/srcdir.dup". To create a duplicated file (say a source file), invoke your favorite editor with the following pathname "/users/fa/srcdir.dup/hello.c". When writing, the file is updated on two repository file systems, as set by configuration parameters. Afterwards, invoke `make` and run your program; even if one of the repository file system fails during the make, the compiler will finish correctly!

This mechanism of request redirection is in fact very similar to I/O redirections or pipes in UNIX. In the above case, the only thing to be careful of, is to respect the protocol between the Process Manager and the File Manager. This protocol is actually part of the UNIX subsystem interface, and thus can be quite easily used.

4.5.3 Static Extensions

There is, in the system description of a process, an "*Extend*" class whose member functions are invoked on process system calls such as `fork(2)`, `exec(2)`, `exit(2)`, allowing system writers to add functionality to UNIX processes, by pure extension of the CHORUS code. This hook has been used to easily implement a CASE/PCTE UNIX Subsystem (on AT/386) on top of the CHORUS Nucleus.

4.6 Examples

4.6.1 Development of the Pipe Manager

In the early versions of the UNIX Subsystem, pipes were implemented within the File Manager (derived from System V code). Since then, pipe management has been extracted from the File Manager and rewritten as a UNIX process which can invoke any UNIX system services.

At init time, a Pipe Manager creates a port and inserts it into a group representing the pipe service. It then enters an infinite loop: waiting for incoming messages carrying requests (e.g., pipe creation, read, write), serving the request, replying to the request. As it uses only IPC to receive and reply to requests, it can be invoked and tested by user programs using the IPC interface of the UNIX Subsystem. This also allows running and testing this new implementation without disturbing the service provided by the running UNIX Subsystem, as pipes services are still provided by the File Manager.

Pipe management does not deal with either traps or interrupts, so the Pipe Manager does not need to be loaded in the system address space. It gets the address space protection of any UNIX process, which makes it easy to debug. Other benefits from having a system server be a UNIX process are that traces may be redirected to a file (using shell mechanisms), a crash of the server does not affect the system as a whole, and standard UNIX debuggers such as `sdb` can be used.

However, once fully tested, the Pipe Manager is relinked and loaded into the system address space, thus avoiding additional memory context switches when it is invoked. Finally, the pipe routine of the Process Manager needs to be modified to invoke the new Pipe Manager instead of the File Manager in case of pipe system calls. Only then does the system needs to be stopped and reloaded with the new version of the Process Manager.

4.6.2 Development of a new file system

Developing a new version of the File Manager follows the same steps than those outlined above for the Pipe Manager, except that disk drivers perform privileged instructions for I/O operations, and symbolic ports can be used to connect the file tree managed by the new File Manager under test to the file tree

managed
by the current operational File Manager.

To access disks from user space, the following mechanism has been used. To be allowed to access privileged instructions a thread must be executed in the system address space. Before loading the File Server being tested, a small process that connects two functions to traps using `svCallConnect` and one function to the disk interrupt using `svConnect`, is loaded into the system address space. One of the trap calls is used to perform privileged I/O instructions to start the I/O. When the driver needs to issue such instructions, it just does the corresponding trap. The other trap call is used to wait for an incoming interrupt, it is used by the thread dedicated to delayed interrupt processing. When this thread starts, it enters an infinite loop: wait for interrupt with the trap function, and trigger the appropriate interrupt routine. When an interrupt occurs, the connected function is activated by the CHORUS Nucleus. This function posts an event which is awaited by the trap function called by the Interrupt Thread.

When the new File Server has been initialized, it creates a symbolic port in the system file tree, say "/tmp/newfs". Each access to files such as "/tmp/newfs/users/fa/myfile" will thus be received and served by the new File Server as an access to the file "/users/fa/myfile". This allows the full testing of the new File Server using standard UNIX utilities.

Of course, the new File Manager needs to be tested either on a machine with two disks or on a machine with one disk, booted as a diskless station, using a remote file system.

To replace the current version of the File Manager with the new one, the system must be stopped and reloaded. Avoiding stopping the system would imply that File Managers are stateless, or that they can transmit their current state to each other, which is somewhat complex to implement.

5. Lessons and open issues

The UNIX Subsystem on CHORUS shows clearly all the benefits one can gain from modularity in operating system development. However, improvements can still be made and alternative solutions to some of the issues raised by such an implementation are worth considering. Some of these are currently being studied in new versions of the system.

5.1 Performances

Regarding performances, modularity is not as expensive as is usually thought. Table 3 summarizes some initial performance measurements done on a COMPAQ 386/25. It compares the UNIX Subsystem of CHORUS with the Microport system.

TABLE 3. – Performance of the UNIX Subsystem

Primitives	**CHORUS**	**Microport**
`getuid`	*85 μs*	*80 μs*
`sbrk(0)`	*95 μs*	*128 μs*
`read (1Kb)`	*146.8 Kb/s*	*107.2 Kb/s*
`write (1Kb)`	*121.2 Kb/s*	*56.6 Kb/s*
`pipe (4096)`	*415 Kb/s*	*1212 Kb/s*
`exec`	*17 ms*	*37 ms*
`fork`	*17 ms*	*26.5 ms*

The read and write tests work on a 2 Megabyte file, 1 Kilobyte at a time. The pipe test writes and reads 4 Kilobytes blocks through a pipe. These measurements illustrate the viability of implementing a system as a set of servers without loss of performance. This topic, which is discussed in Section 5.3 leads to some other performance improvements, which will be illustrated there.

5.2 Modularity

Modularity has proven to be very convenient and powerful to adapt the system to hardware configurations. More modularity could be obtained in particular with the extraction of disk drivers from the File Manager to have them run in separate actors. This would allow File Managers to deal easily with remote disks, thus permitting to access floppy disks of diskless stations without any File Manager running on such a station.

5.3 Servers and Threads

As the UNIX service has been split into independent servers, more system resources are needed in order to be able to respond to client requests than in other UNIX kernel implementations. For example, for every u_thread created in a user process one should create one thread in the File Manager, one in the Device Manager, one in the Socket Manager, etc. Only such a policy can insure that system resources will be numerous enough to respond to client requests. This is especially true for servers in which threads can be blocked for a long or infinite time waiting for an incoming event: read on a terminal for Device Manager for example. While a server thread is blocked, other client requests cannot be served by that thread. If all the threads are blocked for reading, no process can write on terminals any longer! In other words, this means that as modularity increases, resource consumption rises, overloading the Nucleus tables with (most of the time) idle threads.

In fact, it is possible to configure the servers in such a way as to much diminish the problem, although without eliminating it. Mechanisms are being studied to transparently transform Remote Procedure Calls to local routine calls when the destination port is located on the same site than the sender. Thus a server is executed as a "monitor" by the u_threads which issued the system call. As a result, though modularity is preserved, the consumption of system resources is lowered. Threads running in servers only serve incoming remote requests.

Some preliminary developments in that direction have clearly shown its promise. A particular protocol between the Process Manager and the File Manager has been developed to simulate such a behavior, and to permit some real measurements. This transparent transformation of RPC into routine calls impacts the system in two other ways:

- When a process invokes a server, the code of the server runs at the priority of the process. High priority processes can be served with respect to their priority. When a real RPC is performed, the request is performed at a standard priority as defined in the server. Thus, this transformation makes it possible to provide users with a more real-time system.
- Executing server code in the context of the calling thread avoids context switches improving system performance. Some measurements have been done on the system emulating this feature. In this system, transformation of RPC into routine calls has been done for read and write operations. Results are shown in Table 4.

TABLE 4. – Performance of the UNIX Subsystem when converting RPC

Primitives	CHORUS with true RPC	RPC converted to routine call
`read (1Kb)`	*146.8 Kb/s*	*211.2 Kb/s*
`pipe (4096)`	*415 Kb/s*	*1240 Kb/s*

5.4 Caching

The distributed file system provided by the UNIX Subsystem is based on direct access of the client to the appropriate File Manager by means of the CHORUS IPC. This makes it possible to maintain full consistency with UNIX file system semantics.

An important drawback of such a choice is that there is no caching of remote data on the client side. Rather than having File Managers cooperate to cache remote data, the use of virtual memory mechanisms is being studied to implement file access. This still avoids loading a File Manager on a diskless station. Using virtual memory services makes it possible to take advantage of its caching mechanisms. Open files can be manipulated as segments cached by the virtual memory manager but not necessarily mapped into a particular region.

5.5 Symbolic Ports

Symbolic ports allow transparent interconnection of the UNIX name space, and provide a powerful extension mechanism. This is done without any lexical exception in pathnames. But introducing a new file type in the UNIX world implies that some standard utilities (less than 10) must be modified to take this new file type into account, e.g., `fsck(1)`, `cp(1)`, `find(1)`, `test(1)`. In fact, lexical exception is avoided by a semantic exception!

A general mechanism allowing to transparently (either from a lexical and a semantic point of view) connect servers to any node of a UNIX file tree seems more appropriate and convenient. Feasibility of such a mechanism is being investigated.

6. Conclusion

Making the CHORUS Nucleus *generic* prevented the introduction of ‘‘features’’ with ‘‘heavy’’ semantics. For example, features such as application-oriented protocols, fault tolerant strategies, do not appear in the CHORUS Nucleus. However, it provides the building blocks to construct these features inside subsystems.

On the other hand, CHORUS provides effective, high performance solutions to some of the issues known to cause difficult problems to system designers, mainly system (re)configuration (static and dynamic), adaptability, extensibility, and debugging, which is eased by isolating resources within actors and by communicating by means of messages providing explicit and clear interactions.

The CHORUS modular structure has been very successful, allowing to provide binary compatibility with UNIX, while keeping the implementation well structured, portable and efficient.

All these principles were those on which UNIX was initially designed 20 years ago on a standalone time-sharing computer. Networks and multi-processors introduce today new features and constraints that force one to "rethink" the internal structure of UNIX in order that it still be a modern operating system. CHORUS obviously shows that UNIX can be reminded of its original virtues and, while still keeping its standard interface for application programs portability, (r)evolve to the next generation of systems...

7. Acknowledgements

Vadim Abrossimov, Ivan Boule, Hugo Coyote, Corinne Delorme, Jean-Jacques Germond, Lori Grob, Marc Guillemont, Sylvain Langlois, Pierre Léonard, Pierre Lebée, Ian Liang, Jim Lipkis, Marc Maathuis, Denis Metral-Charvet, Will Neuhauser, Maria-Inès Ortega, Bruno Pillard, Didier Poirot, Eric Pouyoul, François Saint-Lu and Eric Valette contributed, each with a particular skill, to the CHORUS-V3 implementation on various machine architectures.

Many thanks are addressed to the referees for their helpful comments, and additional credits to Lori Grob for improving the readability of this paper.

8. References

[Abro89] Vadim Abrossimov, Marc Rozier, and Marc Shapiro, ‘‘Generic Virtual Memory Management for Operating System Kernels,’’ *Submitted for publication*, (April 1989), p. 20.

[Abro89a] Vadim Abrossimov, Marc Rozier, and Michel Gien, ‘‘Virtual Memory Management in Chorus,’’ in *Lecture Notes in Computer Sciences, Workshop on Progress in Distributed Systems and Distributed Systems Management*, Springer-Verlag, Berlin, Germany, (18-19 April 1989), p. 20.

[Acce86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, ‘‘Mach: A New Kernel Foundation for UNIX

Development,’’
in *Proc. of USENIX Summer’86 Conference*, Atlanta, GA, (9-13 June 1986), pp. 93-112.

[Alve88] Jose Alves-Marques, Roland Balter, Vinny Cahill, Paulo Guedes, Neville Harris, Chris Horn, Sacha Krakowiak, Andre Kramer, John Slattery, and Gérard Vandome, ‘‘Implementing the Comandos Architecture,’’ in *Esprit’88: Putting the Technology to Use*, North-Holland Publishing Co., (November 1988), pp. 1140-1157.

[Cher88] David Cheriton, ‘‘The V Distributed System,’’ *Communications of the ACM*, vol. 31, no. 3, (March 1988), pp. 314-333.

[Herr88] Frédéric Herrmann, François Armand, Marc Rozier, Michel Gien, Vadim Abrossimov, Ivan Boule, Marc Guillemont, Pierre Léonard, Sylvain Langlois, and Will Neuhauser, ‘‘CHORUS, a New Technology for Building UNIX Systems,’’ in *Proc. of EUUG Autumn’88 Conference*, EUUG, Cascais, Portugal, (3-7 October 1988), pp. 1-18.

[Li86] Kai Li and Paul Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ in *Proc. of Principles of Distributed Computing (PODC) Symposium*, (1986), pp. 229-239.

[McJo88] Paul R. McJones and Garret F. Swart, ‘‘Evolving the UNIX System Interface to Support Multithreaded Programs,’’ Technical Report 21, DEC Systems Research Center, Palo Alto, CA, (September 1988), p. 100.

[Mino88] Régis Minot, Pierre Courcoureux, Hubert Zimmermann, Jean-Jacques Germond, Paolo Alvari, Vincenzo Ambriola, and Ted Dowling, ‘‘The Spirit of Aphrodite,’’ in *Esprit’88: Putting the Technology to Use*, North-Holland Publishing Co., (November 1988), pp. 519-539.

[Mull87] Sape J. Mullender et al., *The Amoeba Distributed Operating System: Selected Papers 1984 -1987*, CWI Tract No. 41, Amsterdam, Netherlands, (1987), p. 309.

[Pres86] David L. Presotto, ‘‘The Eight Edition UNIX Connection Service,’’ in *Proc. of EUUG Spring’86 Conference*, Florence, Italy, (21-24 April 1986), p. 10.

[Rash87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew, ‘‘Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures,’’ in *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, (October 1987), pp. 31-39.

[Rozi88] 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.

[Tane86] Andrew S. Tanenbaum, Sape J. Mullender, and Robert van Renesse, ‘‘Using Sparse Capabilities in a Distributed Operating System,’’ in *Proc. of IEEE 6th. International Conference on Distributed Computing Systems*, CWI Tract No. 41, Cambridge, MA, (19-23 May 1986), pp. 558-563.

[Wein86] Peter J. Weinberger, ‘‘The Eight Edition Remote Filesystem,’’ in *EUUG Spring’86 Conference*, Florence, Italy, (21-24 April 1986), p. 1.

[Coyo89] Hugo Coyote, ‘‘Spécification et Réalisation d’un Système de Fichiers Fiables pour le Système d’Exploitation Réparti CHORUS,’’ PhD.Thesis, Université Paris VI, Paris, France, (June 1989), p. 300.

CONTENTS

LIST OF FIGURES

LIST OF TABLES