UNIX on a Loosely Coupled Architecture: The CHORUS/MiX Approach

Lawrence Albinson, Dominique Grabas, Pascal Piovesan, Michel Tombroff, Christian Tricot and Hossein Yassaie.

approved by:

abstract: Paper presented at the EIT Workshop on Parallel and Distributed Worstation Systems, September 26-27, 1991, Florence, Italy.

Contents

Abstract

In the CHORUS/MiX® distributed operating system architecture the microkernel provides system servers with generic services which are independent of a particular operating system; these services include processor scheduling, memory management and inter-process communications. In turn, co-operating system servers provide at the application programmer's interface a particular operating system personality. The CHORUS/MiX implementation of UNIX® is based on AT&T source code, but is significantly re-structured into a set of system servers. This re-structuring has resulted in a modular and adaptable system which is well suited to distribution across a loosely coupled parallel architecture.

The CHORUS/MiX system is further being developed to provide what has been termed *single site semantics* (SSS). This will make it possible to create the illusion of UNIX running on a single processor whilst taking advantage of the availability of a number of loosely coupled processors. The IMS T9000® Transputer will be one of the first processors on which CHORUS/MiX SSS will be implemented.

1 Introduction

During the late 80's it became possible to build parallel machines which used shared memory. The rationale for this approach was that individual processors, particularly those with caches, did not fully utilise available memory bandwidth, and fast arbitration could be implemented which would permit several processors connected to the same bus to use the available bandwidth. Various versions of UNIX were developed which exploited, with some success, these kinds of architectures. The parallelism which this provided was mainly exploited by pipe connected processes, background activities such as networking demons, the presence of multiple users, and multiple login sessions through the use of window systems. Although several processors could usually be launched in a single address space, so called multi-threading, few applications were written to exploit this facility. The technical challenge in the software was to devise a reasonably systematic means of interlocking the single processor UNIX kernel code. Of course, deadlock avoidance was an issue, as was identifying and hopefully eliminating locking *hot spots*. However, the fundamental bottleneck was always going to be the physical sharing of a memory bus. Experience showed that performance usually worsened beyond around twenty processors.

During the same period there were significant advances in building machines of many hundreds of processors which interworked not by memory sharing but by high performance communications, and in devising parallel algorithms which were a good match for the concurrency "granularity" which these machines provided. Individual processors lacked any memory protection, machines were typically dedicated to specific applications, and the software environments were usually rather weak and hosted on a more conventional machine such as a workstation.

Recently the trend has been towards highly parallel machines which interwork using communications but which also have memory protection in each processor. This has brought together the UNIX world and the world of dedicated parallel applications. No longer will it be necessary to "host" these highly parallel machines; no longer will it be necessary to have incompatible

® CHORUS and CHORUS/MiX are registered trademarks of Chorus systèmes.
® UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S.A. and other countries.
® IMS T400, IMS T800 and IMS T9000 are registered trademarks of INMOS Limited.

processors in the mainframe and in the workstation. Of course, this uniformity across the performance spectrum comes at a price. The most important questions which must be answered are: How do we make a communications type parallel machine look like a single UNIX machine? How do we permit parallel applications to run on a parallel machine in a way which does not disturb either UNIX or other parallel applications running on a different group of processors in the same machine? Can we dynamically expand and contract the part of a parallel machine which is given over to providing UNIX. These questions are briefly addressed in section 2.

Operating system architecture would ideally be driven by the needs of applications. However, as new machine architectures have emerged, the architecture of operating systems has had to evolve to fit them. At the same time OS architecture has had to change to cope with ever greater complexity. The newest OS's attempt to be as general as possible by assuming the machine model to be a possibly large collection of loosely coupled nodes, where each node is either a mono-processor or a shared memory multi-processor. CHORUS/MiX is one such operating system. In the CHORUS/MiX distributed operating system architecture each node of a communications machine, be it a mono-processor or a multi-processor, runs a small micro-kernel. The services provided by this micro-kernel are designed to be independent of any particular operating system; these services include scheduling, memory management, inter-process communications and inter-node communications. In turn, co-operating system servers provide at the application programmer's interface a particular operating system personality. The CHORUS/MiX implementation is based on AT&T source code, but is significantly re-structured into a set of system servers. This re-structuring has resulted in a modular and adaptable system which is well suited to distribution across a loosely coupled parallel architecture. Increased modularity has also helped to mitigate the effects of significantly increased complexity. In section 3 we give a more in-depth description of the CHORUS/MiX architecture.

A good example of a highly parallel communications machine is the Inmos transputer. The key features of the T400 and T800 families of transputer were fine grain parallelism through lightweight process state and hardware scheduling, and high bandwidth communications through specially designed inter-processor links. The T9000 family of transputer extends these features with the introduction of virtual channels, memory protection and richer trap, exception and interrupt support. The T9000 will be one of the first implementations of a highly parallel CHORUS/MiX. In section 4 we give a brief description of the T9000. We end the paper with a description of some of the issues involved in porting CHORUS/MiX to the T9000.

2 UNIX on a Communication Machine

2.1 The Rationale For Single Site Semantics

Imagine a parallel machine in which each processor runs its own UNIX kernel. You might think of most processors as being diskless workstations with a few, the ones with devices connected, being the input/output sub-system. Instead of the processors communicating using a LAN such as Ethernet, they do so using some special facility, for example the transputer link. What you have is a network which suffers from all the problems you hear about from users of workstations and server configurations. For example, if you want to use several processors you have to log into each of them separately, and you have to know their names; you have access to several machines

through different windows but the file system name space is different on each of them; you can't launch a parallel application onto several processors except by making use of sockets which you find too heavyweight; you can't even arrange for a shell pipeline to execute on several processors except by use of special shells which are not widely available. The answer to these problems is to provide the illusion of a single UNIX machine even though that machine is built from a possibly quite large group of processors. This is known in the jargon as *single system image* or *single site semantics*. It has the virtue of hiding in the kernel all the issues of distribution which include load balancing, maintaining a single process identification space and file name space, device naming, time management, resource accounting, and swap space management.

2.2 Parallel Domains On Parallel Machines

Machines of several hundred processors may provide a balanced configuration for specialised parallel applications. However, it is unlikely that they would be balanced for typical use as a UNIX machine as they would almost certainly lack sufficient input/output capacity. For this reason a single system image UNIX is unlikely to scale much beyond several tens of processors. Indeed there are aspects of UNIX semantics which make further scalability difficult to achieve without a major reconstruction of all the existing kernel code. What is required is to provide UNIX on a small part of a parallel machine together with a method of setting up parallel application domains. Extensions to UNIX will manage the launch, communications and termination of parallel applications. Within a domain a parallel application has available to it the raw performance of its group of processors, unimpeded by the need to maintain UNIX semantics; at the same time it is provided with a means to communicate with the UNIX domain for services such as input/output.

2.3 The Boundary Of A Single System Image

It would be undesirable to fix at bootstrap time the number of processors in a machine given over to running UNIX. This is because the optimum balance of processors to input/output capacity will vary from site to site. Equally, it would be unacceptable to statically fix the number and size of the protection domains. To do so would imply a prior knowledge of the type of parallel applications which will be encountered. This suggests that a single system image UNIX should also offer dynamic reconfiguration. In other words, individual processors can be caused to join or leave the UNIX domain in an orderly way. But if this is possible then why should the boundary of a single system image be a machine? The possibility is there to include each workstation, file server, compute server or supercomputer in the system. The challenge here is to provide a stable view of things when individual machines fail.

3 CHORUS

The CHORUS family of operating systems is centered around the small real-time distributed CHORUS *Nucleus* which provides a set of generic services. The other members of the family are built on top of the Nucleus and inherit its real-time distributed computing capabilities. The CHORUS/MiX operating system incorporates the CHORUS Nucleus as a base on which it builds

a UNIX interface that is transparently extended to distributed processing and to use in real-time environments.

The key characteristics of the CHORUS family of operating systems are:

- *Real-time*: CHORUS systems are built on the real-time CHORUS Nucleus and have all the functional characteristics and performance of classic real-time executives.
- *Controlled transparent distribution* of processing and of data: the management of distribution can be entirely taken care of by CHORUS systems, yet controlled by network administrators. In particular, CHORUS permits dynamically reconfiguring the system and applications.
- *Modularity*: CHORUS systems are composed of a set of communicating modules that can be assembled and reconfigured dynamically depending on the hardware capability and application needs.
- *Openness*: CHORUS/MiX provides all the functionalities of a standard UNIX system (UNIX SVR3.2 today, SVR4.0 in the future).
- *Compatibility* with UNIX is provided by CHORUS/MiX: application programs, utilities, and files used under UNIX are supported without modification (binary compatibility) by CHORUS/MiX.

The immediate domains of application for CHORUS products are those for which the limits of traditional operating systems are preventing the development of fully satisfactory solutions. This includes, for instance, the development of standard operating system on advanced hardware architectures, in particular parallel machines like Networks of Transputers.

3.1 Basic Structure - Nucleus and Subsystems

The CHORUS architecture is based on a small real-time distributed *Nucleus* that integrates distributed processing and communication at the lowest level. The Nucleus provides generic tools - thread scheduling, real-time event handling, network-transparent inter-process communications (IPC) and memory management - for independent servers called *subsystems*, which coexist on top of the Nucleus.

Subsystems separate the functions of the operating system into sets of services provided by autonomous servers, and provide operating system interfaces to application programs.

The CHORUS organisation of Nucleus and subsystems represents the most logical view of an open operating system for cooperative computing environments. Separating functions increases the modularity, portability, scalability and "distributability" of the overall system, which has a small, trustworthy micro-kernel as its foundation.

3.2 The Nucleus

The CHORUS Nucleus (see Figure 1) provides both local and global management of services. At the lowest level, it manages the physical resources at each "node" with four clearly defined

components:

- a classical *real-time multi-tasking executive* which controls allocation of local processors, manages priority-based preemptive scheduling of CHORUS *threads*, and provides primitives for fine grain synchronisation of, and low-level communication between, threads;
- a *distributed memory manager* which can support the full range of memory architectures - linear, segmented or virtual;
- a low level hardware *supervisor* which dynamically dispatches external events such as interrupts, traps and exceptions to dynamically defined routines or ports;
- an *Inter Process Communication* (IPC) manager provides the high-performance global communication services (exchange of *messages* through *ports*) which are the key to CHORUS architecture's distributed capabilities.

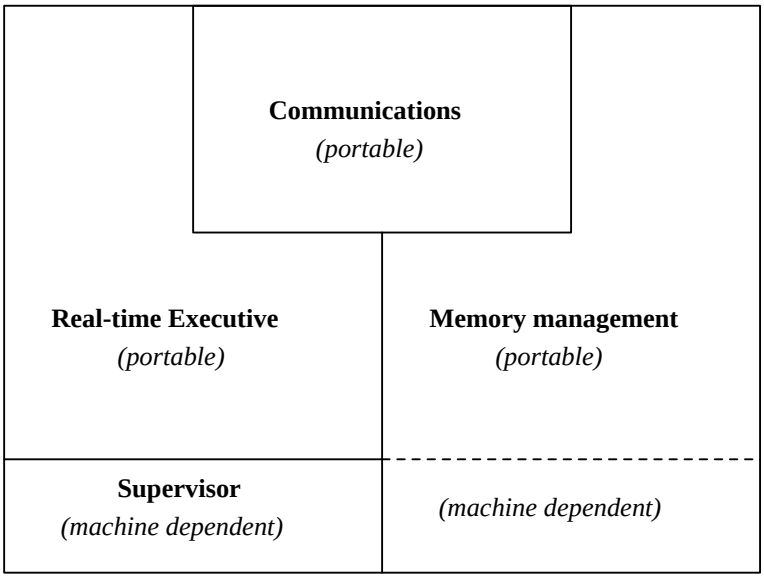

Figure 1: The CHORUS Nucleus

3.3 The CHORUS Subsystem Concept

Subsystems in CHORUS architecture are sets of system servers that use the generic services of the Nucleus to provide higher-level services. More simply, a subsystem is an operating system built on top of the CHORUS Nucleus.

Subsystems manage physical and logical resources such as files, devices and high-level communication services and they communicate via the IPC facility provided by the Nucleus. Their position "on top" of the Nucleus provides a structured, well-defined way for system builders to cope with complexities of operating system development.

In the operating system context, a subsystem offers the means by which to supply a complete standard operating system interface to standard application programs. New servers can then be

added to the set of servers delivering this interface as a means of gracefully extending system and operating system capability. *Servers* are the building blocks, the toolset, that system builders use to incorporate new distributed functions.

3.4 The CHORUS/MiX Subsystem

CHORUS/MiX (see Figure 2) integrates a UNIX System V subsystem with the CHORUS Nucleus to provide a standard-based, real-time, transparently distributed UNIX environment.

CHORUS/MiX servers run on top of the CHORUS Nucleus. Several types of servers may be distinguished within a typical UNIX subsystem: Process Manager (PM), Object Manager (OM), Device Manager (DM) and Socket Manager (SM). The PM maps UNIX processes onto CHORUS abstractions (actor, thread and regions). There is an OM on each site supporting a disk. An OM provides UNIX File System management and acts as a *mapper*, acting as a segment server to the memory management module. A DM is used on a site whenever tty's, pseudo-tty's or bitmaps are connected to that site. The SM is the server which manages internet network protocols such as TCP, UDP and IP accessed through the BSD socket interface.

CHORUS/MiX UNIX services conform to X/Open specifications and have been extended to real-time and to the distributed environment (distribution of programs as well as files), all in a way that is completely transparent to UNIX application programs. A CHORUS/MiX that conforms UNIX System SVR4.0, with real-time and distributed features, is under development.

CHORUS/MiX offers the traditional set of UNIX functions for creating, destroying processes and managing signals, but extends them to manage real-time, multi-threaded and distributed processes. This last extension permits the creation - and manipulation from a distance - of processes on any machine, while respecting rules of UNIX regarding environments, open files and so on.

3.5 Distribution of CHORUS/MiX Servers

3.5.1 Modular Architecture

The subdivision of the CHORUS/MiX operating system into a set of cooperating servers not only provides functional modularity (each server implements a particular set of services), but also *distribution modularity* [Armand et al. 89].

Distribution modularity refers to the possibility of distributing CHORUS/MiX servers across the sites of the network. As CHORUS/MiX servers use the CHORUS Nucleus IPC mechanisms for communicating with each other, they implicitly benefit from all the advantages allowed by the flexibility and transparency of these mechanisms.

The different addressing semantics (functional, broadcast, associative) combined with port grouping and port migration, offer the basic tools for transparently managing the distribution of the CHORUS/MiX servers. Server distribution among the sites of the network may dynamically be modified to adapt to the possibly changing network topology, transient traffic or system load variations.

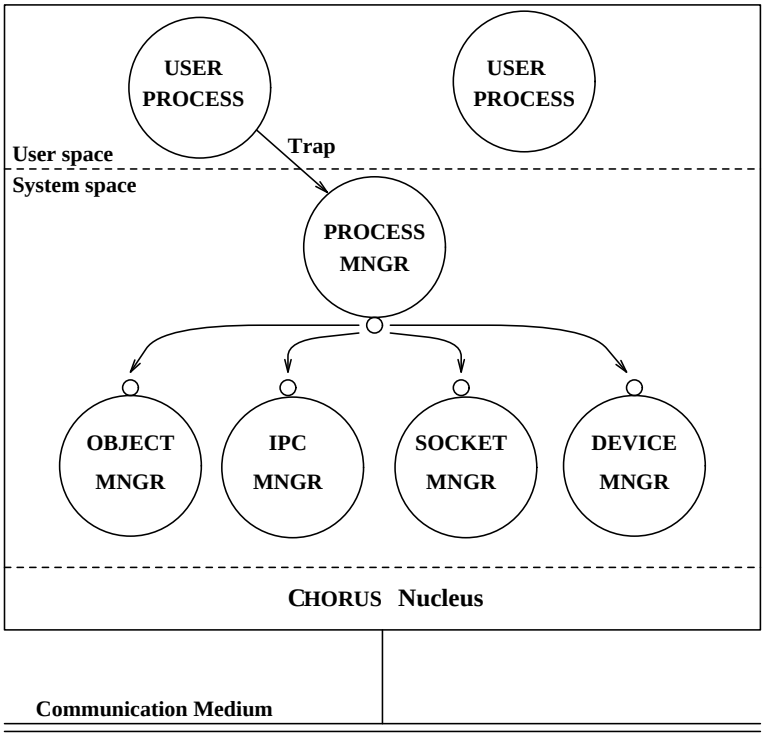

Figure 2: The CHORUS/MiX subsystem

3.5.2 Remote Accesses

As mentioned before, there is an OM on each site supporting a disk. UNIX processes running on diskless site have access to disk abstractions (file systems, home directory, *etc*) by transparently communicating with a distant OM. Note that the PM managing the process does not need to know the actual location of the OM on the network, since it relies on the services of the Nucleus transparent IPC mechanism.

Other servers, like the Socket Manager (SM), do not need to be duplicated on all sites. A SM is typically installed at the site responsible for handling the ETHERNET connection. UNIX process socket communications are serviced locally if the process site has a SM, or by a remote SM accessed transparently by the IPC mechanism.

PMs, which must be present on all "UNIX sites", also provide support for remote operations. A UNIX process may request for the execution of a child process on a remote site (this is known has the remote fork mechanism). It can also migrate from one site to another. In both cases, the Nucleus IPC mechanisms and the global naming paradigm are used by the PMs to manage the distribution (handling of signals, file contexts, etc).

3.5.3 Server Consistency

Duplication and distribution of system servers raise the issue of server consistency.

First, since objects managed by dedicated servers may be accessed concurrently by remote

entities (such an object may typically be a UNIX file or device), it is the responsibility of the server managing the object to guarantee that its consistency is maintained. Multi-threaded servers use locking mechanisms and semaphore operations to synchronise the accesses to the underlying objects.

Second, since servers may be duplicated in order to maintain a high degree of availability and to provide support for fault-tolerance, care must be taken to ensure that services are executed with the required semantics (at-most-once, exactly-once, etc). The servers themselves do rely on the underlying IPC addressing semantics to implement these higher level requirements.

3.6 User Defined Servers

The CHORUS architecture provides a convenient platform for operating system development. The homogeneity of server interfaces provided by the CHORUS IPC allows system builders to develop new servers and integrate them at will into a system, either as user or as system servers. For example, new file management strategies such as real-time file systems, or fault-tolerant servers can be developed and tested as a user level utility without disturbing a running system, using the powerful debugging tools available for user-level application development. Later, the server can be migrated easily within the system for performance consideration.

4 The T9000 Transputer

INMOS Transputers are complete microcomputers integrated in a single VLSI chip. Each Transputer combines an integer ALU, a two-priority hardware scheduler, some on-chip memory, an advanced external memory interface and communication links. In addition, some versions incorporate a floating point unit or some application specific logic. The strength of the Transputer lies in the close integration, both at the micro-architecture and software level, of communication and (multi-)processing. This makes the design of parallel or concurrent systems using Transputers especially easy and efficient.

First generation Transputers, the IMS T400 and T800, did not offer much support for building complete operating systems: no memory protection hardware and no specific trap mechanism. The T9000 is the first member of INMOS' new generation of Transputers. In addition to improved performance, the T9000 has several features that assist in building a distributed UNIX environment [INMOS 91].

4.1 Communicating through Virtual Channels

Communication and parallelism have always been key features of INMOS processors. Mechanisms are offered, at the instruction set level, for building parallel programs of communicating processes. The first generation of Transputers offered intra-Transputer communication through memory to memory DMA and inter-processor communication through DMA over serial communication links. The `input` and `output` instructions gave direct access to the underlying hardware running at 20Mbits per second.

Though enabling efficient parallel machines and programs to be built, this initial design induced undesirable limitations on some application programs.

To lift all these limitations, INMOS introduces virtual channels on the T9000 family of Transputers. A virtual channel enables communication between any processes in a network of processors, irrespective of their physical location. Any number of virtual channels can be handled by a network and each processor can have access to up to 65536 of them.

4.1.1 The T9000's Virtual Channel Processor

The T9000's Virtual Channel Processor (VCP) is the piece of hardware which handles all off-chip communications. Its main role is to multiplex all the outgoing and incoming communications over the chip's four physical links. The VCP makes this multiplexing totally transparent to the CPU. As far as the programmer is concerned, there is a quasi-infinite number of communication channels going off-chip. Which physical link they will use is only known when the VCP is configured, before any application starts executing.

The fact that the VCP is a separate entity from the CPU also means that computation and communication can proceed entirely in parallel, overlapping each other.

To sustain the high communication bandwidth required by the VCP, a new electrical protocol has been devised for the links; the "Data/Strobe" mechanism enables the T9000 to run its links at 100Mbits/s, giving a global bidirectional bandwidth of 80 Mbytes per second.

4.1.2 The IMS C104: a Generic Routing Chip

The T9000's VCP enables any number of channels to link the Transputer with the external world, but this is not enough to allow direct communication between any two processors in a network. This is the job of a specialised chip, the IMS C104 [May and Thomson 90].

The IMS C104 is a general purpose worm-hole packet-switched routing chip. It cooperates with the T9000 in the following way:

- Any message sent by an T9000 over a virtual channel is split into a number of small packets by the VCP. This will enable the VCP to fairly multiplex physical links by multiplexing the packets. Each packet contains a minimal header describing the destination of the message.
- The links of the T9000 being connected to a IMS C104, the packets are routed to their destination encoded in the header. Worm-hole routing means that the routing of a packet takes place as soon as the header has been received by the IMS C104, potentially before the whole message has entered the chip. This mechanism makes routing extremely fast; less than a micro-second is spent between reception of the first bit of a packet and its retransmission out of the IMS C104.
- At the destination side, a C104 delivers the packet to the appropriate T9000 link, where the VCP will reconstruct the message from all the successive packets associated with the appropriate channel.

Providing two separate chips for computing and routing offers many advantages:

- For small networks with only neighbour communications, T9000s can be used without IMS C104, sparing the silicon cost of on-chip routing for other T9000 functionalities.
- Routers can have many links (32 for the IMS C104), minimising the number of routers in a network, and hence minimising the routing delay.
- Since messages do not flow through the T9000s the total link bandwidth required is just the one required by local processes.
- Network structure and scalability is independent of the number of processing nodes.

4.2 Enhanced Process and Scheduler Models

4.2.1 Process Behaviour Control: the Dual Processes

A new type of process, and a new mode of execution, have been added to the features of T9000. The new process type, or *L-process*, offers local handling of error conditions, debugging traps, and a special instruction for implementing operating systems "system calls". The new mode of execution, or *P-mode*, allows an L-process to run in a protected virtual address space.

Both P-mode and L-processes behave in a very similar way. In each case, the execution stream switches to another process (the L-process *stub* for the P-mode, a *trap-handler* for the L-process) whenever an exceptional situation is reached. This other process, or controlling process, deals with the exceptional situation on behalf of its dual. Thereafter, the initial process can be continued or discarded depending on the seriousness of the exceptional situation.

The two main differences between P-mode execution and trap-handling lie in the reduced instruction set available to P-mode (i.e. no scheduling or communication instructions) and the execution of P-mode processes in a segmented and protected virtual address space.

4.2.2 A Rationalised Scheduler

The interface to the scheduler has been cleaned-up and rationalised to allow easy manipulation by real-time kernels or operating systems. The first rationalisation involves the manipulation of an interrupted process, of particular importance to real-time kernels implementing fast interruptible handlers.

When the scheduler swaps from a low-priority to a high-priority process, the state of the interrupted process is saved in a shadow copy of the processor's registers. This copy can be saved to/restored from memory using special instructions. With this mechanism, the T9000 allows clean manipulation of its internal state, enabling improved software scheduling on interrupts.

Another aspect of this improved interface is the availability of atomic instructions to modify the hardware scheduler queues, enabling operating systems and real-time kernels to use the lightningly fast context switching time of the Transputer, while maintaining many more levels of process priority than the two offered by the T9000 hardware.

Finally, semaphores, first introduced by Dijkstra in 1965 as a useful mechanism for controlling access to shared resources by concurrent processes, have been included in the T9000 features. Two new instructions, `signal` and `wait`, have been introduced to atomically handle semaphore structures allocated in memory. This feature greatly simplifies the design of software kernels, as many processes usually need to gain access to shared kernel information.

5 Porting CHORUS onto the T9000

Porting the CHORUS/MiX operating system onto a new hardware architecture is a two-phase process.

First, the Nucleus must be adapted to the new hardware in order to offer the same set of essential generic services to the higher-level components of the system. These generic services are defined by a clear functional Nucleus interface and by a description of the basic abstractions exported by the Nucleus. For keeping the Nucleus highly portable, the hardware dependencies have been isolated in small and well-defined components: the *supervisor* and the *machine dependent memory management module.*

Second, the hardware dependencies of the system servers implementing the UNIX Subsystem must be modified in the same way. As for the Nucleus, system servers have been designed in such a way to allow porting with minimum effort.

In this section, we describe the technical aspects of adapting the CHORUS Nucleus to the T9000 architecture. The basic abstractions implemented and managed by the CHORUS Nucleus are shown in Table 1.

Actor	unit of resource allocation, and memory address space
Thread	unit of sequential execution
Region	unit of structuring an Actor's address space
Segment	unit of data encapsulation

Table 1: CHORUS Nucleus basic abstractions

The *actor* is the unit of resource allocation in the CHORUS system. An actor defines a protected address space (or *context*) supporting the execution of *threads* which share the address space of the actor. An actor is tied to a site, and its threads are executed on that site.

The thread is the unit of execution in CHORUS. A thread is a sequential flow of control and is characterized by a context corresponding to the state of the processor: registers, workspace pointer, instruction pointer, trap-handler identity. A thread is always tied to one and only one actor. The actor constitutes the execution environment of the thread. Within an actor, many threads can be created and can run in parallel. These threads share the resources of that actor.

CHORUS memory management considers the data of a context to be a set of non-overlapping *regions*, which form the valid portion of the context. These regions are mapped to secondary storage objects, called *segments*. Segments are managed outside of the Nucleus, by external servers called segment *mappers*. Concurrent access to a segment is allowed: a given segment may

be mapped into any number of regions, allocated to any number of contexts. The consistency of a segment shared among actors of the same site is guaranteed by the Nucleus, but when a segment is shared among different sites, the segment mapper is in charge of maintaining the segment consistency.

The supervisor contains code and data structures that implement the primary hardware event handling (traps, exceptions and interrupts) and the mapping of threads to hardware resources. It also implements the thread context switch mechanism. Hardware dependencies of the context and region objects are defined and managed by the machine dependent memory management module.

The following sections explain how the CHORUS abstractions are mapped onto the T9000 hardware.

5.1 CHORUS threads and T9000 Processes

CHORUS threads are implemented as independent T9000 *processes*. Threads in *supervisor* mode are mapped onto L-processes (one-to-one mapping) and run without memory protection or address translation. Each of these L-processes is associated with a T9000 *trap-handler* which is responsible for monitoring the traps and exceptions caused by the thread. Trap expense can be avoided for Nucleus system calls executed by supervisor threads, since they execute in the same memory context as the Nucleus itself, and can therefore call the corresponding service routine directly. However, for allowing the possibility of trap nesting and to provide an homogeneity level between user and supervisor threads, the Nucleus handles L-process traps.

Threads in *user* mode are mapped onto P-processes (one-to-one mapping), and run in a protected address space. Each user thread is controlled by a private *stub-process*, which represents the supervisor mode of the thread. The stub is responsible for setting up the memory protection regime, for starting the P-process implementing the thread and for handling the trap and exception events caused by the thread. When an interrupt has preempted a user thread, it is the stub that will restart the interrupted thread when the interrupt handler is finished.

Figure 3 shows the mapping between CHORUS threads (user and supervisor) and T9000 processes.

5.2 Scheduling

Synchronous (de)scheduling, or context switch, refers to the deliberate action performed by the currently running thread in order to de-schedule itself and to schedule another thread.

The Nucleus uses the notion of thread *priority* to implement its real-time scheduling policy. The unique criterium for the processor allocation is priority: at any time, the running thread is the ready thread whose priority is the greatest.

This priority-based scheduling technique is also used to implement the time-slicing mechanism. CHORUS priorities are divided into two sets[1]: threads whose priority stands in the upper

[1]CHORUS priority values range from 0 to 255. Typically, values from 0 to 127 are reserved for real-time threads, and 128 to 255 for time-sliced threads.

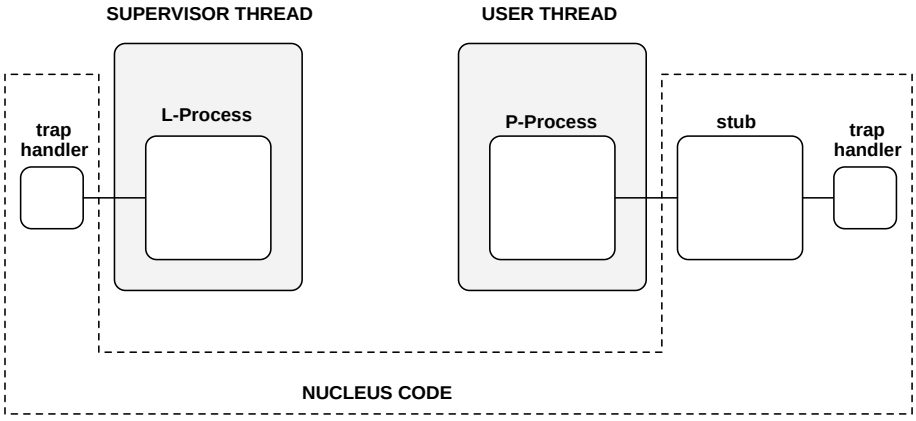

Figure 3: CHORUS threads - T9000 processes

set obey the strict real-time scheduling strategy, while threads in the lower set are subject to a simple time-slicing strategy.

The CHORUS priority is not to be confused with the two-level priority scheme managed by the T9000 hardware scheduler, hereafter referred to as the *low-* or *high-* priority. T9000 processes implementing CHORUS threads are running at low-priority, in order to allow for interrupt preemption. The wider 255-level priority is managed internally by the Nucleus on top of the hardware low-priority.

The context switch mechanism itself has been designed to take advantage of the efficiency of the T9000 hardware capabilities. The problem of implementing the CHORUS micro-kernel real-time executive on top of the T9000 FIFO hardware scheduler is not trivial. Indeed, the CHORUS Nucleus must keep permanent control of the execution sequence, while using at best the benefit of the automatic T9000 hardware scheduler.

The context switch mechanism is implemented using a combination of the T9000 hardware scheduler and semaphore atomic operations. For the purpose of scheduling, each thread is associated with a private T9000 semaphore. This scheme has the advantage that there is no need to save and restore any context information since the T9000 hardware has built-in capabilities for managing the scheduler and semaphore hardware queues. Instruction and stack pointers are automatically updated.

5.3 Hardware Mechanisms: Traps and Exceptions

Traps and exceptions may occur while the currently running thread is in user or supervisor mode. In the former case, the control is passed to the stub process controlling the thread. In the latter case, the trap handler connected to the thread is responsible for servicing the event. The CHORUS Nucleus interface offers a set of routines that allow system servers to dynamically connect trap and exception handler routines. As an example, the Process Manager connects the generic UNIX trap routine at boot time.

A trap is caused by the execution of the T9000 `syscall` instruction. By this mechanism, the thread requests some specific service from the Nucleus. The type of service is passed via

the T9000 integer evaluation stack, and the thread has pushed the arguments on its stack. The supervisor maintains trap routines internally and acts as a dispatcher between the hardware trap mechanism and the actual servicing of the request.

Exceptions are all the events different from traps that may cause the running thread to enter the Nucleus: floating point exception, memory violation, invalid instruction, etc. The action performed by the supervisor in that case depends on the type of exception. As an example, a memory violation may occur when a user thread's stack pointer goes beyond its memory protection range. In that case, the stub extends the stack region and restarts the thread.

5.4 Interrupts

Interrupt sources - timers, event pins and communication links - are monitored by dedicated high-priority processes. These processes, named *vectors*, wait for events to occur on the various sources.

When an interrupt source is triggered, the corresponding vector is awakened and preempts the currently running thread[2]. The vector saves the current thread's context, updates the interrupt nesting level and starts a process (the actual *interrupt handler*) for servicing the interrupt. The vectors and the generic part of the interrupt handlers implement the part of the supervisor not covered by the stub and trap handler processes.

The supervisor is responsible for recording the handlers and vectors that the Nucleus or CHORUS actors have connected to hardware interrupts. It is crucial to have interrupt handlers running at low-priority in order to allow for interrupt nesting and guarantee fast response to concurrent external events.

When the interrupt handler has terminated its job, the preempted thread must be restarted. Before the thread is actually restarted, the Nucleus checks if any scheduling has to be done (a thread could have been made "ready to run" by an interrupt handler). To restart the thread, the supervisor uses the saved context frame. One of the key enhancement of the T9000 compared to the Txxx[3]. is that there is no need to use a high-priority process for restarting the interrupted thread.

Interrupt masking and priority are implemented with a combination of T9000 specific instructions (`intdis` and `intenb`) and T9000 semaphores.

5.5 Memory Management

The CHORUS memory management service provides separate address spaces (contexts) associated with each actor, and efficient and versatile mechanisms for data transfer between contexts, and between secondary storage and contexts. A context is represented by a set of non-overlapping regions, and each region is associated with a segment by the CHORUS memory management module (segments typically represent some form of backing storage, such as the contents of a file).

[2]Recall that threads run as low-priority T9000 processes and can therefore be preempted at any time by high-priority ones.

[3]Txxx refers to the T800 and T400 families of Transputer

The protected mode offered by the T9000 hardware, with its four per-process independent logical address regions[4], is the base on which the region and context abstractions are implemented. In this model, each user actor, if running in protected mode, is assigned four T9000 logical address regions.

The memory management module of the Nucleus must offer a fixed interface to the higher-level parts of the system (region creation and duplication, segment mapping, etc). Because of the T9000 memory architecture characteristics and for efficiency reasons, we have decided to redesign the CHORUS memory management module entirely [Abrossimov et al. 89]. It is based on the same basic abstractions and mechanisms, but the implementation is different. As an example, it was not possible to implement advanced techniques like copy-on-write or -reference. Regions are loaded or copied entirely at creation time.

The unit of memory allocation on the T9000 must be a power of two, and the start address of any allocated memory slot must be a multiple of its size. For these two reasons, the buddy system algorithm has been selected [Tanenbaum 88]. The buddy system has been adapted for speeding up memory allocation in some cases (like for finding small communication buffers). Relocation algorithms are used to maintain a low level of fragmentation, and are implemented on top of the very efficient block move operations allowed by the T9000.

Figure 4 shows the relationship between CHORUS regions and segments with T9000 regions.

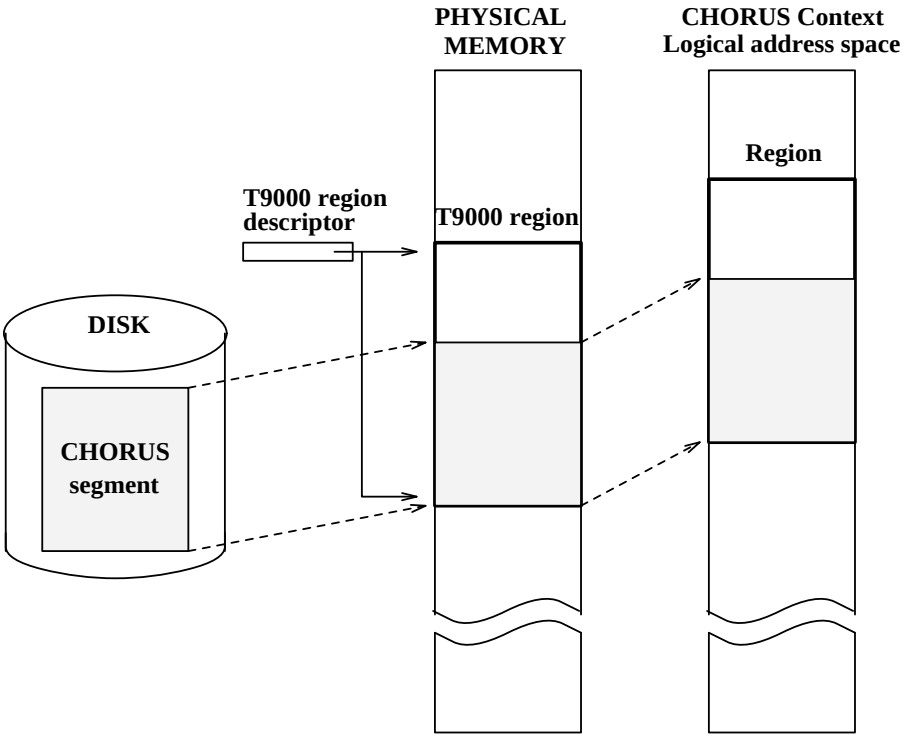

Figure 4: CHORUS segments and regions - T9000 regions

[4]We make the distinction between T9000 region (hardware unit of memory protection) and CHORUS region (contiguous range of logical addresses within a context).

5.6 CHORUS IPC

CHORUS uses communication as the basic function of the operating system. IPC mechanisms must therefore be as efficient as possible and use at best the hardware communication capabilities. The new CHORUS network architecture represents a significant change from earlier network architecture versions. Functionally, few changes are externally visible. The philosophy and implementation of network services is substantially changed, however.

The network protocols have been integrated closely with the CHORUS Nucleus. The CHORUS IPC protocols may be stacked on-top of special purpose transport protocols, offering greater efficiency and providing a consistent interface to non-network devices, such as buses or Transputer links.

The CHORUS network layers are implemented as subroutine layers within a single address space. When possible, user threads to carry out work through the entire depth of the network interface, eliminating inefficiencies involved in copying data and switching contexts between threads or actors.

The major components of the CHORUS network services are shown on Figure 5.

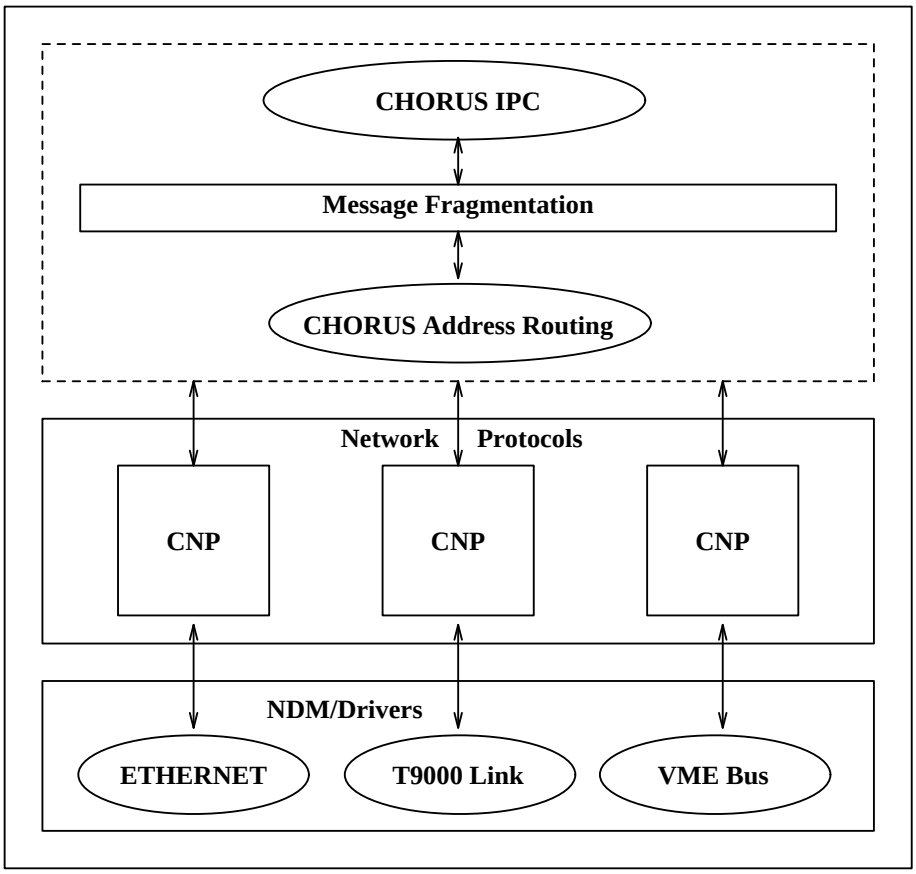

Figure 5: CHORUS IPC Components

- The CHORUS IPC component is responsible for implementing protocols that provide re-

mote IPC and RPC operations for CHORUS ports.

- The *Fragmentation Unit* is responsible for splitting messages into fragments that are conveniently manipulated on output and reassembling them from fragments on input.
- The *Remote Site Manager* implements CHORUS remote site routing algorithms.
- The CHORUS *Network Protocol* component(s) implement an efficient ISO level-3 transport layer.
- Non-CHORUS network protocols may be imported into the CHORUS network framework as *External Network Protocol* components.
- The *Network Device Manager* (NDM) exports a low-level interface to network devices used by CHORUS Network Protocol and external Network Protocol components.

Remote communications are implemented on top of the T9000 Virtual Channel mechanism. Dedicated vectors act as intermediate agents between the NDM and the actual physical links. This design provides fast response for communication primitives and offers a consistent interface to the portable communication module of the Nucleus, which can consider communication links as normal interrupting devices.

Routing and location services at the lowest level as well as packet fragmentation are automatically handled by the T9000 Virtual Channel Processor. More complex routing schemes are also managed by the C104 network at the hardware level. The adaptation of the CHORUS IPC on the T9000 takes advantage of the high-reliability and automatic routing capabilities of the hardware.

5.7 Current Port Status

At the time this paper is written, the CHORUS Nucleus is running on the T9000 simulator: the T9000 simulator is a software package, loaded onto a T805 Transputer, which emulates a T9000 chip. It provides functional simulation, but no timing information. The developments related to CHORUS/MiX (UNIX tools, COFF compiler, *etc*) are also in progress.

We have also adapted the Nucleus for the Txxx family of Transputers, and are currently porting CHORUS IPC on top of Transputer links.

Our research on single system semantics is currently conducted on different types of hardware. CHORUS/MiX V3.2 has been adapted to the iPSC2 Hypercube and is used for experimenting various process migration and load-balancing algorithms. A centralized coherent distributed memory mapper has also been implemented, and its distributed version is now being tested.

6 Conclusion

In this paper, we showed how two state-of-the-art technologies - a micro-kernel based operating system architecture and a high performance communication based processor technology - are combined to build multiprocessor systems supporting a standard operating system.

Naturally, it is expected that these technologies continue to impact and enrich each other over the years to come. There is no doubt that distributed systems and their flexibilities are the core of many emerging application requirements; and that the Transputers and the CHORUS operating system are well placed to be at the forefront of these developments.

One particular area for further development is exploitation of the fault-tolerant capability of the Transputer networks in a CHORUS environment. Provisions will be made for access by applications to load balancing, both in terms of computation and communication, when running under CHORUS. These features, coupled with the time-out capabilities of the Transputer communication mechanism, the possibility of multipath message routing and the process migration paradigm open the way for sophisticated fault-tolerant and reconfigurable systems.

The evolution of the Transputer from its early days, where it offered virtual processing, to the upcoming T9000 with virtual processing, virtual communication capability, and enhanced memory protection/management is expected to continue. In the future the virtual processing and communication capability of the Transputers will be complemented with full virtual memory thus allowing the more sophisticated memory management models offered by CHORUS operating system to be fully exploited.

References

[Cheriton 88] David R. Cheriton, *The V Distributed System*, Communications of the ACM, Vol. 31, No. 3, pages 314-333, March 1988.

[Accetta et al. 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tavanian, and Michael Young, *Mach: A New Kernel Foundation for Unix Development*, USENIX 1986 Summer Conference Proceedings. USENIX Association, June 1986.

[INMOS 91] *The T9000 Transputer Products Overview Manual*, First Edition, INMOS Limited, 1991.

[May and Thomson 90] David May and Peter Thompson, *Transputer and Routers: components for concurrent machines*, Proceedings of the 3rd Transputer/Occam International Conference, Tokyo, Japan, 17-18 May 1990.

[Dyson 90] Clive Dyson, *INMOS H1 architecture revealed*, New Electronics, September 1990, pages 21–24.

[Dettmer 90] Roger Dettmer, *Great Expectations, the H1 Transputer*, IEE Review, December 1990.

[Tanenbaum 88] Andrew S. Tanenbaum, *Operating Systems. Design and Implementation* Prenctice-Hall Software Series, 1988.

[Armand et al. 86] François Armand, Michel Gien, Marc Guillemont, and Pierre Léonard, *Towards a Distributed UNIX System - The CHORUS Approach*, Proc. of EUUG Autumn 1986 Conference, Manchester, 1986.

[Abrossimov et al. 89] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. *Generic Virtual Memory Management for Operating System Kernels*,Proceedings of the 12th ACM Symposium on Operating System Principles, Litchfield Park, AZ, 3-6 December 1989.

[Armand et al. 89] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. *Revolution 89, or Distributing UNIX Brings it Back to its Original Virtues*, Proceedings of Workshop on Experiences with Building (and Multiprocessor) Systems, pages 153–174. Ft. Lauderdale, FL, 5-6 October 1989.

[Armand et al. 90] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier, *Multi-threaded Processes in Chorus/MIX*, Proceedings of EUUG Spring' 90 Conference, Munich, Germany, 23-27 April 1990, pages 1–13.

[Rozier et al. 88] Marc Rozier, Will Neuhauser, and Michel Gien. *Scalability in Distributed Real-Time Operating Systems*. IEEE, Washington, DC, May 1988.