Experience with SVR4 Over CHORUS

Nariman Batlivala, Barry Gleeson, Jim Hamrick,
Scott Lurndal, Darren Price, James Soddy
Unisys Corporation, San Jose, California

Vadim Abrossimov
Chorus Systèmes, Paris, France

Abstract

While many have touted the advantages of microkernel operating system architecture, there are few examples of complete, commercial quality systems built in this style. This paper describes our experience building a UNIX® System V Release 4 (SVR4) compatible operating system using the CHORUS® microkernel.

For this technology to gain acceptance in the marketplace, it must provide full binary compatibility with existing operating systems, and performance comparable to traditional, monolithic operating systems. In addition, it must allow developers to track the evolution of UNIX at a cost not much greater than a port, and it must enable the development of new system functionality.

1. Introduction

Microkernel technology is attractive for many reasons. Until recently, however, investigation of this technology was largely confined to academic settings. Wide acceptance in the commercial world requires solutions not only to the abstract problem of providing appropriate operating system services, but also to the practical problems faced by system vendors each day.

1.1 Requirements

Any commercial version of the UNIX operating system developed today must provide absolute conformance to the source compatibility standards and total binary compatibility with other platforms using the same processor. Since we initially chose the Motorola 88000 as our hardware platform, we have a stringent set of compatibility standards and compliance tests with which to conform. The 88open Binary Compatibility Standard (BCS) defines the trap interface for applications based upon SVR3.2 interfaces, and the 88open Application Binary Interface (ABI) defines the interface for those applications based on SVR4. Price/performance is another key issue. We cannot afford an operating system that prevents us from competing in this area. At the same time, we are prepared to absorb some performance overhead, particularly in non-critical areas, to ease the development of new features and functionality.

The ability to track the evolution of UNIX is at least as important as providing binary compatibility and competitive price/performance. It appears that UNIX is becoming more modular. It is commonplace today for I/O devices to be designed for industry standard busses, and for the device drivers to be provided by the manufacturer of the I/O device. Thus, source compatibility with the **Device Driver**

® UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S.A. and other countries.

® CHORUS is a registered trademark of Chorus systèmes.

Interface (DDI), enabling the direct importation of third party device drivers and STREAMS modules is a key requirement. The **Virtual File System** (VFS) [Kleiman 86] interface has enabled the development of new file systems independent of the rest of the UNIX kernel. Hence, adherence to the VFS interface is also important.

We must be able to track future UNIX System Laboratories (USL) releases such as SVR4 ES and ES/MP, with a cost not significantly greater than that of a simple porting effort. This is a major challenge!

We also have an internal goal: this operating system should be an appropriate base for experimentation with and development of distributed systems. During design and development, we devoted considerable thought to the problem of presenting a ''Single System Image'' to users of a collection of machines running this operating system.

The target hardware for this operating system is the Unisys S/8400, a 33 MHz Motorola 88000-based machine with one or two processors, and a VME based I/O subsystem.

1.2 CHORUS Concepts

The abstractions provided by the CHORUS microkernel or **Nucleus** are similar to those found in other systems, such as Amoeba [Tanenbaum 90], Mach [Golub 90], and the V Kernel [Cheriton 90]. These abstractions include

- **Unique Identifiers**: global names used to provide a unified name space for all objects within a distributed CHORUS system. Unique identifiers (UIs) are guaranteed to be unique over time for all sites within an administratively designated domain.
- **Actors**: collections of resources within a CHORUS site, representing private memory contexts that may contain multiple memory regions, communication ports and threads of execution.
- **Threads**: sequential flows of control within an actor. All threads of an actor share all resources of the actor. Threads are scheduled independently by the Nucleus. Threads synchronize with each other using Nucleus provided **mutex** and **semaphore** operations.
- **Ports**: location transparent communication end-points within a CHORUS system. (A CHORUS system consists of a set of CHORUS sites.) Threads send and receive messages on ports, which serve as globally-named message queues. Ports are initially attached to a specified actor but can be migrated to another actor. Only threads within the actor to which the port is attached have the right to receive messages on it. Knowledge of a port's name implies the right to send messages to that port.
- **Port Groups**: collections of ports that are addressed as a group to perform communication operations. CHORUS port groups provide several addressing modes. Messages may be sent to all members of the group, any one member of the group, one member of the group residing on a particular site, or one member of the group residing on the same site as another known port.
- **Messages**: untyped sequences of bytes that represent information to be sent from one port to another via CHORUS IPC. All messages transmitted are stamped with the protection identifiers of the sending actor and port.
- **Regions**: contiguous ranges of valid virtual addresses within an actor. Each region is treated as a unit by the CHORUS virtual memory system. Multiple regions within a single actor may not overlap.
- **Segments**: encapsulated data within a CHORUS system, typically representing some form of backing store, such as a swap area on a disk. Regions map a portion of a segment into the address space of an actor. Requests to read or modify data within a region are converted by the virtual memory system into read or modify requests within the segment through the use of a standard Nucleus-to-mapper protocol. Any server adhering to this protocol can act as a **mapper** for virtual memory segments. Mappers also provide the synchronization needed to implement distributed shared

memory.

- **Capabilities**: unique handles, the possession of which grants the right to perform an operation. Within a CHORUS system, capabilities consist of the concatenation of a port name and a key. The port name identifies a server and the key identifies an object within the server.

Of the above abstractions, unique identifiers, actors, threads, ports, port groups, messages, and regions are defined and managed solely by the CHORUS Nucleus. Capabilities and segments are managed cooperatively by the Nucleus and system servers. For further information concerning the CHORUS Nucleus abstractions, see [Rozier 88].

A distinguishing characteristic of the CHORUS Nucleus is that the actor abstraction was extended by the introduction of the **supervisor actor.** Supervisor actors are much the same as other actors in that they are compiled, linked, loaded into memory, and unloaded from memory separately from the Nucleus and from each other. They differ from other actors because, when loaded, they execute in a privileged instruction mode and share the system address space.

Supervisor actors enable several optimizations that greatly enhance the performance achievable in a modular subsystem. The two most notable optimizations are the ability to use **connected handlers** and an extremely fast lightweight RPC mechanism.

Connected handlers allow efficient treatment of hardware events. Using an architecture-independent interface, supervisor actors may associate functions dynamically with traps, exceptions and interrupts. The use of connected handlers eliminates the requirement that device drivers be part of the microkernel, without sacrificing performance.

The CHORUS lightweight RPC mechanism (LWRPC) optimizes communication between supervisor actors that reside on a single site. When a LWRPC can be performed, the cost of communicating between the two supervisor actors is slightly more than that of a subroutine call. This optimization is performed automatically by the Nucleus, transparent to the client of the service.

Using these abstractions, a high-level operating system interface, such as that provided by UNIX, is exported. Such an operating system interface is called a **subsystem**. Typically, subsystems consist of several multithreaded system servers, each providing a part of the exported interface. The protocols between the system servers are designed so that the operating system interface can be readily extended to a distributed environment. Each remote request to a server is processed by one of its threads. Part of the request message contains information needed to construct the context within which to process the request.

Each server creates one or more ports to which remote clients send requests. Some of these ports may be inserted into port groups with well-known names. Such port groups can be used to access a service independent of the provider of the service.

Servers may also provide a trap interface to services. These interfaces can be compatible with existing UNIX-like system interfaces, but do not extend transparently across the network. To provide a distributed system interface while maintaining binary compatibility, system calls are normally handled by a server that exports a trap interface. System calls that reference remote entities are marshaled into messages and sent using CHORUS IPC to the server that manages the remote entity.

1.3 Architecture of SVR4 over CHORUS

The operating system consists of up to five distinct servers, depending upon the needs of a particular site. (See Figure 1.) These subsystem servers are

- Process Manager (PM) - manages process contexts and dispatches system calls.
- Object Manager (OM) - manages file systems, block device drivers and non-STREAMS character drivers.
- STREAMS Manager (STM) - manages STREAMS drivers, pipes and all other STREAMS dependent services.
- IPC Manager (IPCM) and IPC Key Manager (KM) - manage System V IPC (shared memory, semaphores and message queues) and their associated keys, respectively.

The servers are known collectively as the **SSU**.

This division into servers is similar to that used in CHORUS/MiXV3.2, a version of UNIX System V 3.2. The IPCM and KM extend the operating system services to include all System V IPC facilities. The Device Manager of MiX V3.2 was replaced in the SVR4 implementation by the STM. In addition, file data is cached by the Virtual Memory subsystem, which is managed by the Nucleus. This required the development of significant new protocols between the OM, PM and Nucleus.

The OM, STM, IPCM and KM use large sections of unmodified SVR4 code, and are written in C. The Nucleus and PM are written in C++.

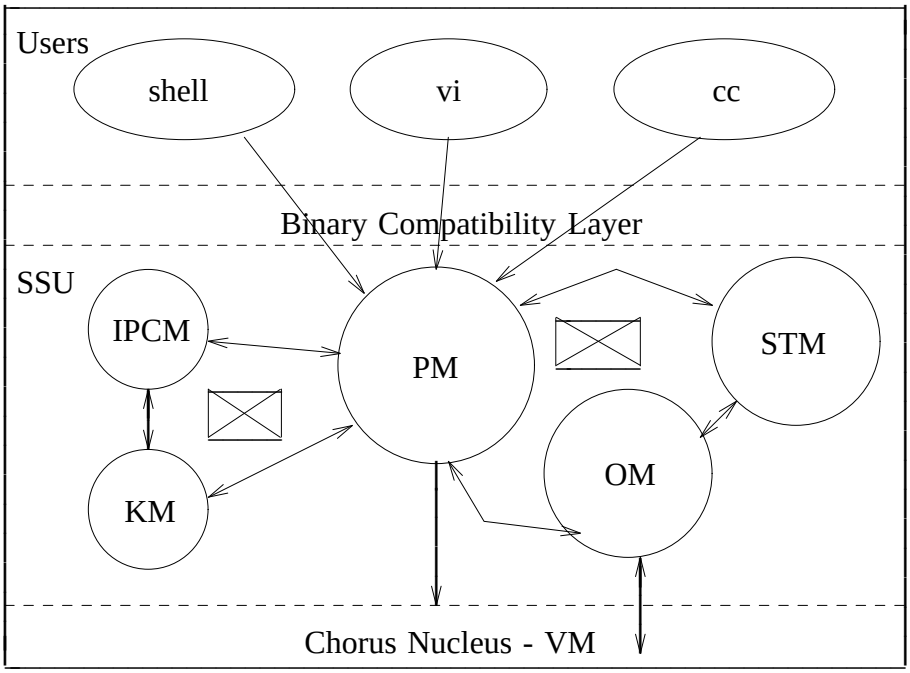

Figure 1. SVR4 Servers

2. File System

SVR4 introduced new file system services, and a major file system reimplementation. Among the new services are *mmap*(2) and enforcement mode record locking, which pose some interesting problems in a distributed environment, such as that provided by the CHORUS Nucleus. Since *mmap*(2) permits simultaneous access to file data through the load and store operations, as well as the through traditional *read*(2) and *write*(2) system calls, the USL implementation features elaborate interactions between the file system code and the Virtual Memory code.

The CHORUS model calls for a separation between the client Nucleus (or user process) and the mapper. Implementing this separation means that the Nucleus and the OM do not share data structures. We describe here the mechanism we have chosen to provide the services and semantics of SVR4 files while maintaining the model of client and server separation.

2.1 Memory Mapped Files and the Page Cache

A typical UNIX system call for file system services begins as a trap into the PM. In the case of *open*(2), a message is sent to an appropriate OM determined by the pathname of the file. If the open is for a cacheable (mappable) file, the open will return a special **capability** which is used to locate the mapper (OM) and identify the file to the mapper. The following sections describe the how reading and writing data is done through the Nucleus page cache.

2.1.1 Read. When a *read*(2) system call is initiated on a cacheable file, a *sgRead*(K) call is made to the Nucleus. The arguments include the capability mentioned above. Using this capability, the Nucleus can send an appropriate message to the OM for that file.

In the simplest case, the Nucleus is asked for data that is already available in the page cache. In this case the data is provided with no file system interaction. If the Nucleus is asked for data that is not already cached, a **PULLIN** message is sent to the OM.

If the file is mapped, the PM creates a region in the process backed by the file. A Nucleus call sets up the address space information and provides the capability obtained when the file was opened. A fault for read in this process is handled much as the read cases described previously. Specifically, if the page is not already cached, a PULLIN message identical to the one previously mentioned is sent to the mapper.

Note that the role of the OM for *mmap*() is trivial. The OM must know that mappings exist, because they represent a reference to the vnode and because they must be synchronized with mandatory locking. So minimally, the OM must be told when the first mapping is created, and when the last one is destroyed.

2.1.2 Write. Write is handled similarly to read, but we must consider also the mechanism for maintaining the atomicity and serializability expected for UNIX file system operations. When a *write*(2) system call is initiated on a cacheable file, a *sgWrite*(K) call is made to the Nucleus. As before, the capability required to send a mapper message is included.

Consider the case that the write is for exactly one full page of bytes for a new file. The Nucleus has no requirement to obtain data from the mapper, but the file system must allocate disk space for the data to be written. In this case, a **GETACCESS** message is sent to the mapper asking for **write access** for the correct offset and size. This gives the mapper the opportunity to allocate disk space or to fail the request.

If the request is successful, and the file is not truncated before the next *sync*(2), the Nucleus eventually sends a **PUSHOUT** message to the OM with the new data.

Next consider the case that some bytes are modified in the middle of an existing file by the write. If the Nucleus has the corresponding page with write access, no interaction with the file system is necessary. If the Nucleus has the page with **read-only access**, the Nucleus must (as before) send a GETACCESS to the mapper asking for write access. This gives the mapper a slightly different opportunity than in the previous case. The mapper can determine that no other clients have read access to the page before granting write access to the requesting Nucleus. If the Nucleus does not have the page, it sends a PULLIN message to ask for the data with write access.

Note: In the read case above, the PULLIN message carries with it a request for read access to the

relevant portion of the file. In the event of read/write activity on a file, the mapper similarly determines that there is no other client with write access, before granting the read access.

If a write fault occurs for a page of the same file, the interactions between the Nucleus and the mapper are the same as when the page is modified by a *write*(2) call.

2.1.3 Larger Reads and Writes. Consider the case of a read that spans 100 pages of file data. The Nucleus might be better off to execute a number of small PULLIN requests rather than one large request. I/O atomicity is maintained because the Nucleus may request, for example, the first two pages of data, along with 100 pages of read access with the first PULLIN message of the transaction. The Nucleus holds the read access until all 100 pages have been transferred. This eliminates the possibility of another client obtaining write access during the course of satisfying the *sgRead*(K). A write might similarly involve a large GETACCESS request.

It is an important aspect of this architecture that once access rights are held by a Nucleus for a portion of the file, the I/O may proceed without holding any locks on the file system. This permits us to perform simultaneous I/O for multiple processes on non-overlapping ranges within a file.

2.1.4 Flushing Data and Access Rights. The mapper may need to change the access that a client Nucleus has to all or part of a file. For example, if a file is truncated we must destroy all cached pages. This is accomplished with the *sgFlush*(K) call, which has options to destroy parts of a cache, to downgrade access from write to read, or simply to cause dirty pages to be flushed to backing store.

2.2 Attributes and Tokens

Since file I/O can take place ‘‘behind the back’’ of the file system, a mechanism is needed to keep attributes, such as file size, access time and modification time, current. The system call code in the PM is required to maintain these attributes, if they change as a result of a *sgRead*(K) or *sgWrite*(K) nucleus call. It is further necessary for the PM to ‘‘know’’ the size for such files, so that it can validate the size and offset of a *read*(2), for example. Thus, the PM maintains a client-side attribute cache.

We have implemented a shared reader, single writer token protocol for attribute mangement. The current state of the token is associated with the **vnode** for each cacheable file. When an OM receives the first open request for a file from a PM, the reply message will, if possible, indicate that the PM is receiving the readable token and file size as part of the reply. At the completion of the open, the OM and PM may both have an accurate size for the file.

If the PM is going to change the size of a file, it must first request the writable token from the OM. After granting the request, the OM may no longer know the exact size of the file. The OM, of course, always knows the amount of disk space allocated for use by the file. At sync time, the attributes that should go to disk are sent to the OM in the sync message.

If the OM needs to use or modify the size of the file, for example because of truncation, the OM must recall the appropriate token from the PM that holds it. While the readable token is not present, the PM may not use the size of the file.

The distribution of size information results in an inconvenience for file systems that have blocks size smaller than the system page size. When a small file is created and written, only the PM knows the exact size of the file. The Nucleus asks for write access to the first page of the file, and backing store must be allocated for the entire page. If blocks are allocated but not written, they must be freed when the vnode becomes inactive. In fact, the excess blocks are cleaned up on the last close of a file.

2.3 Mandatory Record Locking

The I/O path is direct from the PM to the Nucleus. The PM must, therefore enforce record locks. When the OM receives a request to place an enforcement mode lock on part of a file, each PM that has

the file open must agree that the lock is in place before the OM can grant the request.

2.4 The Proxy File System

The implementation of STREAMS-based device files is separated into multiple servers. The name space for these files is managed by the OM, while the devices themselves (with their associated drivers) are managed by the STM. This distribution is managed by a pseudo-file system, called the **proxy** file system.

While the PM-Nucleus-OM implementation of files with cacheable data is based on a model of client-side attribute caching in the PM, the proxy file system is based on a model of server-side attribute caching in the STM. Server-side attribute caching minimizes both complexity and messaging overhead for files whose name space management and data management are distributed across actors.

Proxy file system synchronization is controlled by the OM, but the file objects themselves (and their attributes) are managed by the STM. The proxy file system emulates the SVR4 **specfs** file system on the coordinator (OM) side, and emulates the underlying disk-based file system on the server (STM) side. The proxy server-side attribute caching model is implemented by using a coordinator-server protocol between the OM and STM.

When, during the course of a name space lookup operation, the OM recognizes the first open of a file object managed by the STM, the file object attributes are migrated to the STM with a **MOVEINODE** message, which creates a capability for the STM server file object. The OM proxy file system constructs a data structure (similar to a specfs vnode) to which the lookup operation resolves.

An open operation performed on the proxy file object sends a **LEAFOPEN** message to the server STM designated by the server capability. (The conceptual separation of the MOVEINODE operation and the LEAFOPEN operation is necessary to maintain control of the synchronization of opens and closes at the coordinator.) To complete the open operation, the coordinator (OM) returns a capability for the server (STM) file object to the client (PM). Subsequent operations on the open file exchange messages directly between the client (PM) and server (STM) with no intervention by the coordinator (OM).

Operations performed using the server file capability can be handled with a single message from client (PM) to server (STM). Using the server-side attribute caching model, implicit attribute modifications (those occurring as side-effects, such as modification time updates) can be handled at the server. Operations that access the file through the file system name space are still managed at the coordinator (OM). Since the coordinator's copy of the file attributes is not guaranteed to be up to date, an operation such as *stat*(2) requires a **GETATTR** message from the coordinator (where the name resolves) to the server to retrieve the most recent copy of the file attributes. Similarly, an operation such as *chmod*(2) (which also locates the file through the file system name space) requires a **SETATTR** message from the coordinator to the server to modify the attributes cached by the server. When a *sync*(2) occurs, the server (STM) locates all locally-managed file objects with modified attributes, and syncs the attribute information back to the coordinator (OM).

When the last reference to a server file object is removed, the server (STM) sends a **LEAFCLOSE** message to the coordinator (OM). Depending on the sequencing of opens and closes, the coordinator responds with either a **LASTCLOSE** (no other references outstanding) or with another LEAFOPEN (indicating that an open was in progress at the coordinator while the server was handling the close). The server responds to a LASTCLOSE message by releasing any local data structures for the server file object, and returning any modified attribute information to the coordinator.

The server-side caching model minimizes message traffic for operations that access or modify file attributes. Operations like *read*(2) and *write*(2) have the side-effect of updating timestamps for the file. Operations such as *fchmod*(2) explicitly modify attributes that, if the client-side caching model were

used, would not be cached at the client. Operations such as *fstat*(2) access all attributes of a file. If the client-side caching model were used, some operations at the server would be required to fetch attributes from both the client and the coordinator. The server-side attribute caching model provides all the required functionality and distributed synchronization at minimum message overhead in the general case (with only slightly increased cost in the exceptional cases).

3. STREAMS and Device Management

3.1 STREAMS

We attempted to change the SVR4 STREAMS code as little as possible when integrating it into the CHORUS environment. To accomplish this, it was necessary to provide an environment for this code that included constructs such as a **user area**, a **proc structure**, and a **STREAMS scheduler** since such constructs do not exist in the Nucleus itself. In our implementation of UNIX over CHORUS, therefore, messages sent from the other actors to the STM must first execute a **wrapper** layer of code that creates an emulation of the necessary environment before the SVR4 STREAMS code is executed. The information necessary to create the environment (PID of the process making the request, job control disposition of that process, size of a read or write, etc.), is generally contained in the request message itself.

Some elements of the monolithic UNIX environment are too costly to send to the STM in every request message. (Process **credentials** are one such element.) To avoid this overhead, such pieces of the UNIX environment are generally cached in the STM and kept in a consistent state via protocols run between the STM and other actors. For example, although process credentials are actually created and modified in the PM, some are also cached in a read−only state in the STM. In each request message sent to the STM that requires process credentials, the PM instead passes a **credentials capability** that references those credentials. Upon receiving the request message, the STM uses this capability to search its cache of process credentials. If the credentials are not in the cache, the STM sends a request to the PM whose UI is stored in the credentials capability, asking for a copy of the credentials corresponding to the capability. (The cache size is limited by discarding cache entries when a predetermined threshold of unreferenced credentials is reached.) Thus, sending large data structures in every request message is avoided.

3.2 Scheduling and Interrupt Handling

Threads that enter the kernel as a result of executing system calls (as opposed to threads that do so as a result of an interrupt) are commonly referred to as **base level threads**, and are said to be running code at base level. As in previous versions of UNIX from USL, SVR4 does not allow preemption of processes running code at base level in the kernel, although there are **preemption points** in the kernel where processes can explicitly allow themselves to be preempted. Thus, SVR4 is essentially single−threaded at base level in the kernel. In contrast, threads executing in CHORUS actors may be preempted at arbitrary points in their execution. To provide the SVR4 code with the illusion of non−preemptability, the OM and the STM each manage their own **base level mutex**. Before running any kernel code in these actors from base level, the base level mutex must be acquired. The mutex is released when a thread leaves an actor, or when the thread invokes a DDI function such as *sleep*(D3DK) which explicitly allows for rescheduling. Thus, although both the OM and the STM preserve SVR4's single−threading at base level, it is possible for a thread in the OM and a different thread in the STM to be executing concurrently.

Since our operating system is DDI compliant, the SVR4 ***spl*** calls are supported. The mechanism used does not directly manipulate a hardware mask, however. *Spl* calls (such as *splhi*(D3D), *splx*(D3D)) allow base and interrupt threads in SVR4 to synchronize access to critical regions of code. In our operating system, calls to raise the processor *spl* level result in the acquisition of an actor's **spl mutex**. Lowering the *spl* level to 0 results in the release of the actor's *spl* mutex. Interrupts that are directed by the Nucleus to an actor first execute code which attempts to acquire the *spl* mutex. If successful, the

driver interrupt handler executes immediately. If the *spl* mutex is not available, a request is queued for a **deferred interrupt thread** to acquire the mutex and run the interrupt handler later when the mutex is released. Thus, mutual exclusion between base and interrupt level threads accessing *spl* protected code regions is maintained.

3.3 Device Management

In SVR4, the Device Driver Interface (**DDI**) was introduced in an attempt to better define and isolate the interface between the kernel and device drivers. SVR4 provided some backward compatibility for drivers not written to conform to the DDI, but since we were developing an operating system for a new computer architecture and wanted to limit device driver access to kernel data structures, we decided to mandate DDI conformance for all drivers used with our operating system. DDI conformance helps limit the amount of SVR4 environment emulation required of each of the actors.

For the majority of the DDI functions, we used the SVR4 code without change. The *datamsg*(D3DK), *putnext*(D3DK), and *qreply*(D3DK) routines are examples of functions that fell into this category. DDI functions related to virtual memory management (such as *vtop*(D3D)) were modified to interface to the Nucleus primitives, but were otherwise not difficult to port. For DDI functions related to synchronization between base and interrupt level threads (such as *splstr*(D3D), *splx*(D3D), *sleep*(D3DK) and *wakeup*(D3DK)), we chose not to use the SVR4 code, but rather to implement equivalent functionality with new code. In the case of *splstr*(D3D) and *splx*(D3D), this decision was driven by our desire not to have a global hardware mask, so that the OM and STM could handle interrupts independently and not interfere with one another. There were no Nucleus calls to do *sleep*(D3DK) and *wakeup*(D3DK) directly, so we implemented these functions using the Nucleus semaphore primitive[1]. The *physiock*(D3D) function, which sets up DMA between a device and a user's address space, was somewhat difficult to implement primarily because of the finite message size provided by the Nucleus. There were finally a few DDI functions (notably *copyin*(D3DK) and *copyout*(D3DK)) whose interface specification was actively hostile toward allowing them to function in a distributed environment. Some of the problems encountered porting these functions are discussed later in this paper.

4. Process Management

4.1 System Call Handling

The user's interface to the operating system is provided by two mechanisms. Applications compliant to the 88open BCS use trap instructions compiled into the executable file to invoke services from the operating system. Applications compliant to the 88open ABI link at run-time with a library that provides services. By using a trap interface from the library itself rather than the application, the ABI provides a greater degree of application binary portability while ensuring maximum flexibility for the operating systems system call implementation. Among the responsibilities of the PM is the handling of system call trap instructions.

The PM, as part of its initialization, requests the nucleus to establish **trap handlers** for the traps managed by the PM. When a thread executes a trap instruction to a managed trap vector, the nucleus executes the PM trap handler providing the trap number and the context describing the thread at the time of the trap. A value in a register at the time of the trap indexes into a vector containing function pointers for each system call.

Subsequent handling of a system call depends upon which subsystem actor provides the required service. Some system calls, such as *getpid*(2), can be serviced directly by the thread executing the trap handler.

---

1. Francois Armand of CHORUS provided our prototype implementation of *sleep*(D3DK) and *wakeup*(D3DK).

Most calls, however, require service from another actor or another thread within the PM. These calls can be roughly assigned to two functional areas: file system interfaces, such as *open*(2), *read*(2), *ioctl*(2), *mmap*(2), and process control interfaces, such as signal management, process creation and deletion, and retrieving process information. The file system interfaces result in a message being dispatched to the OM or STM for subsequent handling, while process control messages are sent to the control port (see Section 4.2 "Session and Process Group Management") of the target process.

4.2 Session and Process Group Management

SVR4 introduced the notion of process affiliations. A set of processes represent a user's login **session**; those processes may be subdivided into distinct **process groups**. Mechanisms have been provided to manage these affiliations. The user can create process groups that may be assigned as jobs, suspend and resume process groups, and move process groups into the foreground or background of the session's controlling terminal.

We have emulated SVR4 process affiliations and their management using unique identifiers (UIs), ports and port groups. Each process has a **control port** on which it receives various process control messages. For example, signals are sent to processes by sending a message to their control port.

SVR4 process identifiers (PIDs) are mapped to the UI of the process's control port by using *uiBuild*(K). Arguments to *uiBuild*(K) include the PID of the process and the UI type: K_UIPORT. Thus, servers such as the PM, OM and STM can fabricate the UI of a process's control port given its PID. Knowing the port UI then gives a thread the right to send messages to that port.

Similarly, we have mapped the notions of session and process group onto the Nucleus notion of port group. Processes become members of a session or a process group by inserting their control port into the port group for that session or the process group. Port group names, represented as capabilities, are also built by using the *uiBuild*(K) function.

The following examples illustrate the application of the ideas explained above:

4.2.1 Creating Sessions. Upon the successful completion of a *setsid*(2) system call, the calling process (a) is the session leader of a new session, (b) is the process group leader of a new process group, and (c) has no controlling terminal. The session ID and the process group ID are set to the PID of the calling process. Further, the calling process is the only process in the new session and the only process in the new process group.

To enforce these requirements, the following actions are taken:

- A port group UI for the process group is constructed from the calling process's PID, by using *uiBuild*(K). A message is broadcast to all members of the port group, excluding the calling process. If a process replies, thereby indicating that the port group (process group) already has a member, the session creation is not allowed and *setsid*(2) returns an error to the caller.

  If the *ipcCall*(K) returns a K_EUNKNOWN error, thereby indicating that the port group has no reachable member, session creation proceeds.

- Since the process is changing its process group and session affiliation, the process's control port must be removed from the current port groups and inserted into new port groups.

- The process's parent is notified of the change in affiliation by sending the parent a message on its control port.

4.2.2 Creating Process Groups. The system call *setpgid*(2) may be used to create a new process group within the session of the calling process. This requires the removal of the process from an existing process group and installation in a new process group. This is achieved by moving the process's control

port from one port group to another.

Since process affiliations are changing within the session of the calling process, additional work is required to fix up the process group parent linkages. A process group parent inserts its control port into a port group named after the PGID of its child's process group. The process group parent port group also serves as an efficient mechanism for handling orphan process groups. This has greatly simplified the algorithms used in the distributed environment.

4.2.3 Joining Process Groups. The system call *setpgid*(2) may also be used to join an existing process group. This is a cooperative effort between the calling process and some member of the process group. The calling process requests membership in the process group by broadcasting to the target process group, using functional mode to address any single member of the group. Once the request is granted, the calling process inserts its control port into the appropriate port group. Finally, the calling process informs its parent of the process group change by sending a message to the parent's control port.

4.3 Architecture of the `/proc` File System

Two actors are involved in `/proc` file system operations. The OM manages the file system mount point, virtual file system semantics, and open requests for files in the `/proc` file system. The PM acts as a server for the file system (much as the STM acts as a server for STREAMS-based files) handling requests such as *read*(2), *write*(2), *stat*(2), *ioctl*(2) and *close*(2).

Most operations against files in the `/proc` file system are handled in whole or in part by the PM. For example, the *open*(2) system call is sent initially to the OM for pathname analysis; when the `/proc` mount point is encountered by the OM during pathname resolution, a message is sent to the PM to resolve the rest of the path. The PM creates a local data structure associated with the file and returns a capability describing it to the OM. This capability includes the UI for the PM request port so that any further system calls against the file are routed directly to the PM.

4.3.1 Directory Operations The `/proc` file system is mounted at a file system mount point and as such may be the target of standard directory operations such as *opendir*(3), *readdir*(3), and so forth. This directory is constructed dynamically when a directory file is opened and read or when the vnode operation *VOP_READDIR*() is performed on behalf of an *open*(2) call.

4.3.2 Role of the Object Manager Pathname resolution is a function of the OM. Since `/proc` is implemented as a virtual file system, it uses standard pathname resolution facilities. This requires a vnode operations vector for `/proc` with an entry point known as *VOP_LOOKUP*(). When this entry point is called during pathname resolution, the OM will send a message to the PM. The PM searches its internal process list to resolve the leaf name.

A more common directory operation in the `/proc` file system is directory access via *readdir*(3). This involves opening the directory represented by the path `/proc`. Reads and writes on this file must be satisfied by the PM.

When a request to read the `/proc` directory is received, the OM forwards the request to the PM to obtain a buffer of device independent directory entries (in the format prescribed by the *X/Open Portability Guide* definition for `<dirent.h>`). The OM then copies the entries back to the requestor as appropriate to the size of the requestor's buffer and the current offset within the directory (as maintained by the standard offset management facility).

4.3.3 Role of the Process Manager The PM, as previously stated, acts as a server for `/proc` file system requests. The PM service code handles four types of requests: open/close operations, read/write operations, directory operations and ioctl operations.

Service requests to the PM are handled asynchronously with respect to the execution of the thread being managed, yet most requests are allowed only when the target process is stopped. To facilitate this, when

a debugger requests a process stop (via an *ioctl*(2) of **PIOCSTOP**), the PM service thread handling the request first suspends the target thread. It then determines whether the thread is currently executing user mode code, or executing within the operating system processing a system call. In the former case, the thread is left suspended until the debugger explicitly causes it to resume execution (via **PIOCRUN**); in the latter case, a thread-local flag is set to cause the thread to stop itself just before it returns from the system call (in the PM trap handler).

Read and write operations are handled by using the Nucleus primitive *vmCopy*(K) to copy data to/from the stopped thread's address space.

Various control operations may be targeted to the thread by using the *ioctl*(2) system call. Some of the operations include retrieving process credentials, region mappings, *ps*(1) command data, registers, and signal actions. Control operations are also available to cause the target thread to stop on particular signal(s), fault(s) and/or system calls. When one of these events occurs, the thread stops itself and notifies the debugger by using a semaphore that it is stopped.

5. Experience

5.1 Serverization as a Development Aid

Microkernel-based software systems are generally built as collections of servers. The messaging interface between servers (or between microkernel and server) cleanly encapsulates the environmental data used by each operation in a transaction-like manner.

There is generally a crosstalk-induced limit on the number of developers that can effectively be applied to a single software subsystem. By reducing the grain of the subsystem from the entire operating system to an individual server, the overall size of the development team can essentially be increased proportionately to the number of server-level system components, with a corresponding decrease in elapsed development time.

If the messaging interface is sufficiently well-defined (as in, for example, the interface between the Nucleus Virtual Memory and the external mapper), development on the two ends of the interface may proceed independently. In our experience, such parallel development may involve large geographical separation with no significant loss of development time or effort. While it can be claimed that a similar level of interface definition and independent development can be achieved even in monolithic systems, there has historically been a substantial gap between what is theoretically possible and what is commonly practiced.

Our experience with the serverization of SVR4 has highlighted numerous modularity violations in the monolithic implementation. Some features of the operating system (such as controlling terminals, NFS, and /proc) required substantial rework to achieve a clean serverized implementation. The microkernel model lends itself naturally to the common practice of well-defined interfaces between system components and the parallel development of such components.

Functionality testing can often be reduced to verifying the messaging interface: the message request contains all of the relevant parameters and the message reply contains all of the relevant results. The server cannot reference global data structures outside the local address space, and the operation of the server can therefore be unaffected by such external data, nor can the external data be affected by the server.

Given that all of the state relevant to the execution of a particular operation is contained within the server and the message, it has been straightforward to implement server-specific debugging functionality as extensions of the general operating system debugging facilities. Like the implementation of the servers themselves, the debugging extensions are not visible to, nor do they have any effect upon, any

data or state external to the server.

The microkernel interfaces are equally available to servers executing either in system mode or user mode. It is therefore feasible to verify a large part of the functional correctness of a server using user program development tools, given some small amount of emulation of the system environment. Assuming that the compile-and-execute cycle for the system-level development environment involves system reboots, user-level server development presents a considerable time savings. In particular, the provision for user-level external mappers was found to be a significant benefit.

5.2 Copyin/Copyout Emulation

Copyin(D3DK) and *copyout*(D3DK) are DDI functions responsible for moving data from/to the kernel address space to/from the user address space. Each of these functions takes three parameters: a user address, a kernel address and the length in bytes of the data to transfer. Implicit in the semantics associated with the invocation of these two functions is the assumption that there are two address spaces that can be implicitly located without passing any information through the interface to the function itself, namely, the address space of the current running user, and kernel address space. In monolithic SVR4, this assumption is, of course, met.

Our operating system, however, must be concerned with more than just the kernel and user address spaces. Each actor conceptually has its own address space. (In truth, supervisor actors located on the same site share the kernel address space for that site, but this optimization is not visible to them.) Additionally, our distribution goals required us to consider the possibility that a user process on one site might access a device managed by an actor located on another site. In this environment, a more explicit means of identifying the user address space in question was needed. **Actor capabilities** are provided by the Nucleus to name address spaces unambiguously. Thus, it was only necessary to make this information available to *copyin*(D3DK) and *copyout*(D3DK). Unfortunately, our strict DDI conformance constraints made the functional interface immutable, and so we chose to pass the actor capability to these functions via the emulated user area. In each request message sent to either the OM or STM that might result in either of these functions being called, the actor capability for the user process associated with the request is also sent. The **wrapper code** in the OM/STM puts the actor capability in the user area, where it can later be retrieved.

5.3 DMA Operations

Allowing DMA to be performed by devices to/from user memory is fairly straightforward in monolithic SVR4. (The DDI function *physiock*(D3D) is used by drivers to perform these DMA operations.) Both device and target user address space are located on the same machine. Since we wanted to design an operating system that could be distributed, however, we had to devise a means of supporting DMA in a way that did not require the device and the target user space to be co−located. We also had to take into account the finite message size provided by the Nucleus when implementing our solution. (The limit is approximately 64K in the version we run.) Finally, we did not want the solution that we implemented for distributed DMA to prevent the local DMA from working efficiently.

The two conditions that determine the mechanism used to accomplish a DMA operation in our operating system are

- Whether or not the DMA device and the target user address space are co−located.
- Whether or not the amount of data to be moved exceeds the maximum CHORUS message size.

In all request messages sent to the OM/STM that could result in DMA taking place (normally only messages sent as a result of a *read*(2) or *write*(2) system call), sufficient information is provided to allow the above two conditions to be determined. The co−location condition is determined by placing the **actor capability** for the target user actor in the request message, and having the *physiock*(D3D) function check this UI to see if it has been locally created or not.[2] Whether or not the amount of data to

be moved exceeds the maximum message size is determined by putting the user buffer size for the *read*(2) or *write*(2) call into the request message for *physiock*(D3D) to examine.

If the DMA device and the target user address space are co−located, we perform the DMA operation in essentially the same way that SVR4 does, by locking down the user address range. If they are not co−located, then the action taken depends upon the DMA operation size. If it is smaller than the maximum message size, the request or reply message body is used as the DMA target buffer, depending on whether data is being transferred to or from the device, respectively. If the DMA operation size is greater than the maximum message size, a temporary buffer is allocated on the site that contains the DMA device, and the data is transferred between the user address space and the buffer using the *vmCopy*(K) Nucleus call, which allows arbitrary amounts of data to be moved between any two address spaces.[3] The transfer between the buffer and the DMA device takes place in the normal manner. Thus, we can support drivers that perform DMA, even though the device and the target address space are not necessarily co−located.

5.4 Controlling Terminals

The implementation of controlling terminals in monolithic SVR4 uses code in several normally disparate subsystems:

- The process management subsystem identifies session leaders that are allowed to allocate controlling terminals.
- The STREAMS subsystem determines which devices are terminals, and implements arbitration for a session's controlling terminal once it has been allocated.
- The controlling terminal itself is accessed by opening the special character driver associated with the name `/dev/tty` in the file system name space.

These subsystems access each other's data frequently. While this is not a problem in monolithic SVR4, it causes trouble in our implementation. Process management data structures are located in the PM, STREAMS data structures are located in the STM and the driver for `/dev/tty` is most naturally located in the OM. Thus, direct access of one subsystem's data by another is not possible in our operating system; messages must be exchanged between actors instead. Since we divided the SVR4 code for the various subsystems into separate actors, we were forced to invent a new mechanism to implement controlling terminals. We also wanted to have a mechanism that could be used in a distributed environment, and that would perform reasonably well.

In our operating system, controlling terminals are implemented using a **controlling terminal port group**, which is a port group containing the port of the STM that currently contains the controlling terminal. There is one such port group per session. The port group's UI is created via the *uiBuild*(K) Nucleus call from the session ID of the session that allocates the controlling terminal. When this allocation takes place, the STM that manages the terminal inserts its port into the controlling terminal port group for the session. The `/dev/tty` driver can then access the controlling terminal for a session by converting the session ID of the calling process into the controlling terminal port group UI, and sending a message to the STM whose port is contained in that group. Access to a controlling terminal can be terminated by removing the STM's port from the controlling terminal port group, so that further attempts at communication fail for lack of a reachable destination port.

---

2. Actor capabilities have information about the site where the actor originated embedded in their UIs. CHORUS provides a function (*uiIsLocal*(K)) to determine if a UI is local to a particular site.
3. CHORUS is considering removing the finite message size constraint from their message interface. This would eliminate the need for the *vmCopy*(K).

5.5 Memory Allocation

SVR4 introduced dynamic memory allocation in the kernel via the *kmem_alloc*(D3DK) DDI function. This memory allocator takes large, virtually contiguous chunks of memory and fragments them into sizes useful to the various subsystems of the kernel. Since the SVR4 kernel is monolithic, there is only one such allocator in the system. In our operating system, however, we have multiple system actors, each (conceptually) with its own address space. The Nucleus has a memory allocator built into it (*rgnAllocate*(K)), but that allocator operates at the page level of granularity. Since the memory overhead caused by internal fragmentation would be prohibitive using this allocator, we decided to implement *kmem_alloc*(D3DK) in the actors, and use *rgnAllocate*(K) to allocate the large chunks of virtually contiguous memory necessary for *kmem_alloc*(D3DK) to function.

Although we were initially compelled to use a separate memory allocator for each actor because of the lack of a suitable global allocator in the Nucleus, this decision had benefits that we did not fully appreciate until we began to debug our operating system. Memory leaks (buffers allocated but never freed) were easier to localize than they would have been in monolithic SVR4, since we had information about memory allocation on a per-actor basis. Corruption of dynamically allocated memory was also easier to localize, since either a driver or other code in the actor that saw the corruption was almost always responsible for the problem.

One might wonder what the cost of having a separate memory allocator for each actor is. In our operating system, the size of the implementation of *kmem_alloc*(D3DK) and its cohorts, *kmem_zalloc*(D3DK) and *kmem_free*(D3DK), is about 15K of text, 5K of data, 2K of bss and 12K of data structures dynamically allocated during initialization, for a grand total of about 34K per actor. Considering the amount of memory required to run UNIX on most machines, we think that the cost of multiple memory allocators is reasonable and is outweighed by the benefits they provide.

5.6 Experiences with Process Management

Several race conditions were exposed by the multithreaded implementation of the PM. For example, care was required when a parent and child call *exit*(2) simultaneously, to prevent a zombie child from remaining in the system indefinitely.

In SVR4 a process hierarchy is maintained with the use of linked lists and pointers. In our implementation, an exiting parent process communicates with its child process, and vice versa, via messages. If multiple processes in the heirarchy are exiting simultaneously, it is hard to notify successive generations about their next-of-kin.

5.7 Experiences with /proc

The original SVR4 design and implementation of the `/proc` file system was constructed using a monolithic kernel implementation. The process data structures (proc structure and u area) were accessible directly from the virtual file system code. In our implementation, the process class structures are contained within the PM and the virtual file system structures are contained within the OM and therefore are not shared as in traditional monolithic UNIX implementations.

This led to a division of the `/proc` code between the two actors, with the OM handling file system operations and passing what it cannot handle on to the PM. Because of the multithreaded nature of our implementation, the guarantees and assurances provided by a single-threaded monolithic UNIX implementation taken advantage of by the existing `/proc` code could not be provided. This caused some interesting synchronization problems between the target processes and the debugging process.

Requests handled by the PM to control a target process were delivered to the PM request port. This port is serviced by several threads that execute asynchronously (and preemptively) to other PM threads and user threads. This required the use of some thread state and a pair of semaphores (with mutexes for

protection) to coordinate between the target thread and the PM service thread performing some operation on the target thread.

Another interesting semantic of the `/proc` file system is the ability for a process to exit while debuggers are still accessing the process. Since the open file capability used for files in the `/proc` file system contains the process id of the target process, that PID can be used as a search key on the PM process directory to ensure that once a process exits, successive file system calls targeting that process return an error indication to the caller. A side-effect of the search operation if the process exists is to obtain a lock on the process that prevents exit and signal operations from completing until the `/proc` operation is complete.

5.8 History Object Garbage Collection

The CHORUS paged virtual memory model uses **history objects** to defer data copies until they are required [Abrossimov 89]. In the event that a process forks, the child's data segment becomes a leaf of the binary tree rooted at the parent's data segment. After a cache miss in the child, pages are found by searching upward toward the root of the tree. As pages are modified at the root of the tree, copies are moved down to the history object.

As noted in the Abrossimov paper, if a process repeatedly forks and exits, it can leave a growing chain of history objects. We hoped not to see a problem here with ''real world'' applications. As it turns out, at least some large multi-user benchmark programs exhibit exactly the described behavior. Hence, garbage collection of history objects was added to the paged virtual memory model.

5.9 File System Porting Issues

A great deal of time and effort went into designing and implementing the interfaces that support the client-server model. The monolithic SVR4 VM/file system interface is an elaborate arrangement of coroutines, recursive locks and special cases (e.g., the pageout daemon). By contrast, the microkernel/external mapper model provides a small number of well-defined interfaces for the movement of data between the Nucleus and the file systems.

Our current implementation eliminates some vnode operations and adds some new ones. Those eliminated as a result of our Nucleus interface changes are *VOP_GETPAGE*(), *VOP_PUTPAGE*() and *VOP_MAP*(). Operations added are *VOP_PULLIN*(), *VOP_PUSHOUT*() and *VOP_GETACCESS*(). These interfaces support the external mapper services described in section 2.1.

We added one vnode operation for the purpose of attribute synchronization: *VOP_OBJSYNC*(). It is used to adjust time and size information in the file system specific inode. This operation may also free any allocated, but not used (section 2.2), disk space.

To support attribute distribution, tokens must occasionally be recalled from clients before an operation can proceed. An example is file truncation. We have implemented generic token services that are flexible and usable by any of the file system types.

The *rdwri*() code is never exercised for regular files, which go through the PULLIN, PUSHOUT interfaces. For directories and symbolic links, we use the buffer cache. Therefore, we were able to eliminate all knowledge of Nucleus and address space data structures from the file systems.

We have ported the **s5**, **ufs** and **nfs** file systems from the SVR4 distribution to this environment. Most functions from the porting base either survive intact or are removed completely. The new functions that must be added and other specific changes are very similar between file systems. At this time, porting a new disk based file system to this interface should be little more than a ''cookbook'' effort.

5.10 DDI Compliance of SVVS

Since USL introduced the DDI as the standard interface between device drivers and the kernel in SVR4, and since we had no existing non−DDI compliant drivers for our architecture, we saw no reason to to support non−compliant drivers. We did not anticipate, however, that USL would ship a version of SVVS for SVR4 that incorporated non−DDI compliant STREAMS drivers! We (eventually) got a waiver for the sections of SVVS that required the use of these drivers, so that our operating system could be certified as being SVID 3 compliant.

Primitive Instruction Counts	
Operation	*Instruction Count*
ipcCall[call]	200
ipcCall[reply]	100
User trap to first PM instruction	77
Return to user from PM	123
Interrupt to first OM/STM instruction	151
mutexGet, no contention	5
mutexRel, no contention	9
SemV, no waiters	102
SemV, waiters, no switch	293
SemV, context switch	612
SemP, no contention	97
SemP, context switch	612

Table 1.

6. Performance

The ideal performance comparison would, of course, be to run identical benchmarks on identical hardware, varying only the operating system. We do not have that luxury: we do not have a standard implementation of SVR4 running on the S/8400. However, we do have some performance measurements.

6.1 Micro Measurements

We have instrumented our hardware to enable us to collect instruction and address traces of arbitrary length while the system is running. The measurement technique has some drawbacks: timing of external events is somewhat distorted, and the traces include all delayed branch slot instructions, whether or not they are executed. (The Motorola 88000 is a RISC processor, which always fetches the instruction immediately following a branch. The instruction may or may not be executed, depending on the branch instruction type, even if the branch is taken.) The instruction counts for various Nucleus calls are included in Table 1.

Contended mutex operations have the same cost as contended semaphore operations. The cost of traps and basic synchronization operations are comparable to those of non-microkernel operating systems. *ipcCall(K)* costs are real overhead, compared to operating systems that cannot be distributed.

At this point, little effort has been expended to optimize these costs.

6.2 Macro Measurements

Some macro benchmarking and optimization have been done on this operating system. However, the insights gained from the instruction tracing work described above have not yet been incorporated into the system. Hence, the numbers quoted here should be regarded as initial values only.

The single processor S/8400, with 32 KB of instruction cache, 32 KB of data cache, 64 MB of memory, and one disk, delivers 204 AIM III user loads.

7. Conclusion

We met the basic requirements of binary compatibility and reasonable price/performance relatively easily. System performance at this point is competitive, but not yet outstanding. Compliance with the DDI interface is complete — importing new device drivers and STREAMS modules is straightforward. Importing new disk-based file systems is essentially a cookbook operation.

It is still too early to judge how easy it will be to track new USL releases.

The system is robust and quite stable — we have been using it in-house as our development platform now for over a year, and it has been shipping to customers for more than six months. Extending it to run on the dual processor hardware was simple: essentially no changes were required outside the Nucleus. Experimentation with distribution has begun, and the preliminary results are encouraging.

With several years of development experience using this technology, we are convinced that this is an appropriate way to build commercial operating systems.

8. Acknowledgements

Without the dedication and hard work of the following people, this project would never have seen the light of day: Francois Armand, Chris Bertin, Philip Chan, Yu Chang, Jan-Hua Chu, Gilles Courcoux, Joyce Crowley, Jean-loup Gailly, Jean-Jacques Germond, Gordon Harris, Frederic Herrmann, Billy Ho, Michael Ho, Srivatsan Kasturi, Sandy Lee, Herman Li, Sam Li, Fong-Ching Liaw, Linda Lin, Luke Lin, Jim Lipkis, Lisa Liu, Susan Lor, Jarrett Lu, David Lyle, Anup Pal, Eric Pouyoul, Ellen Reier, Mark Rooke, Marc Rozier, Bob Schatz, Mike Scheer, Graham Stott, Virendra Vase, Bhaskar Vissa, Cheng-Mu Wen, Tim Williams, Becky Wong and Kwame Yeboah.

9. References

[Abrossimov 89] V. Abrossimov, M. Rozier, M. Shapiro. "Generic Virtual Memory Management for Operating System Kernels." Proc. of 12th ACM Symposium on Operating Systems Principles, Litchfield Park, Arizona (USA), December 1989, pp. 27

[Bershad 90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. "Lightweight Remote Procedure Call." ACM Transactions on Computer Systems, vol 8, 1, February 1990, pp. 37-55

[Cheriton 90] David R. Cheriton, Gregory R. Whitehead, Edward W. Sznyter. "Binary Emulation of UNIX using the V Kernel." Proc. of Summer 1990 USENIX Conference, Anaheim, California (USA), June 1990, pp. 73-86

[Golub 90] Davic Golub, Randall Dean, Alessandro Forin, Richard Rashid. "UNIX as an Application Program." Proc. of Summer 1990 USENIX Conference, Anaheim, California (USA), June 1990, pp. 87-96

[Guillemont 91] Marc Guillemont, Jim Lipkis, Douglas Orr, Marc Rozier. "A Second Generation Micro-Kernel Based UNIX: Lessons in Performance and Compatibility" Proc. of the Winter 1991 USENIX Conference, Dallas, Texas (USA), January 1991, pp. 13-22

[Kleiman 86] S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." Proc. of the Summer 1986 USENIX Conference, Atlanta, Georgia.

[Rozier 88] Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, Will Neuhauser. "CHORUS Distributed Operating Systems." Computing Systems Journal, vol 1, 4, The Usenix Association, December 1988, pp. 305-370

[Tanenbaum 90] Andrew Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, Guido van Rossum. "Experiences with the Amoeba Distributed Operating System." Communications of the ACM, volume 33, number 12, December 1990, pp. 46-63

[Williams 89] Tim Williams, "Session Management in System V Release 4." Proceedings of the Winter 1989 Usenix Conference.