1990

# An Overview of the Virtual Memory Xinu Project

Douglas E. Comer
*Purdue University*, comer@cs.purdue.edu

James Griffioen

Report Number:
90-1028

# AN OVERVIEW OF THE VIRTUAL MEMORY XINU PROJECT

Douglas Comer
James Griffioen

CSD-TR-1028
October 1990

# An Overview of the Virtual Memory Xinu Project

*Douglas Comer*
*James Griffioen*

*Department of Computer Science*
*Purdue University*
*West Lafayette, IN*
*47906*

October 3, 1990

CSD-TR-1028

*ABSTRACT*

The Virtual Memory Xinu Project investigates a new model of virtual memory in which dedicated, large-memory machines serve as backing store (page servers) for virtual memory systems operating on a set of (heterogeneous) clients. The dedicated page server allows sharing of the large physical memory resource and provides fast access to data.

This paper gives a brief overview of the Virtual Memory Xinu research project. It outlines the new virtual memory model used, the project's goals, a prototype design and implementation, and experimental results obtained from the prototype.

## 1. Background

Virtual memory operating systems provide mechanisms that allow programs requiring large amounts of memory to execute on a wide range of architectures regardless of the computer's physical memory size. This ability aids the development of portable code by allowing programmers to design and implement programs independent of the physical memory size available on the underlying machine. When user programs exhaust all available local physical memory, the operating system writes blocks of physical memory to backing storage. The virtual memory system later retrieves the blocks of memory from the disk on demand.

Most conventional virtual memory operating systems use magnetic disks for backing storage[8]. Magnetic disks provide high data transfer rates, large storage capacity, and the ability to randomly access data, making them an appealing backing storage medium. The operating system usually reserves a fixed size region of the disk for backing storage and writes blocks of data directly to that region (i.e. no

additional file structure or other organizational structure is imposed on the raw storage provided by the disk)[1,5,6].

More recent virtual memory systems have added a level of abstraction to the paging paradigm. These systems use the abstraction of files to hide the underlying storage device from the virtual memory system and allow the operating system to store data on the disk using high level file operations. The virtual memory system does not need to know the characteristics or organization of the underlying disk device because the file system handles the storage of data on the disk. Some operating systems provide support for a distributed file system and allow diskless machines to use remote files for backing storage[7,10,11]. Unfortunately, using files for backing storage increases the overhead associated with paging. Writing data to a file usually requires a minimum of 2 disks accesses (1 or more to update the directory structure and 1 to write the data) whereas writing directly to the disk requires only 1 access[1,5]. Moreover, file systems often attempt to improve performance with techniques such as *read-ahead*. Read-ahead assumes that most programs access files sequentially and attempts to prefetch additional data whenever the system reads from a file. However, when applied to random access paging activity, prefetching wastes valuable buffer space, degrading both paging and file system performance.

The Virtual Memory Xinu project explores a new model of virtual memory in which dedicated, large-memory machines serve as backing storage for virtual memory systems operating on a set of (heterogeneous) clients. The dedicated memory server† allows sharing of the large physical memory resource and provides fast access to data.

This paper gives a brief description of the Virtual Memory Xinu project. It describes the model used and outlines the system components and their design.

## 2. The Remote Memory Model

The Virtual Memory Xinu project uses a new model for virtual memory called the *remote memory model*[4]. The remote memory model consists of several client machines, one or more dedicated machines called remote memory servers (or page servers), and a communication channel interconnecting all the machines. Figure 1 illustrates the architecture.

† We use the terms *memory server* and *page server* interchangably.

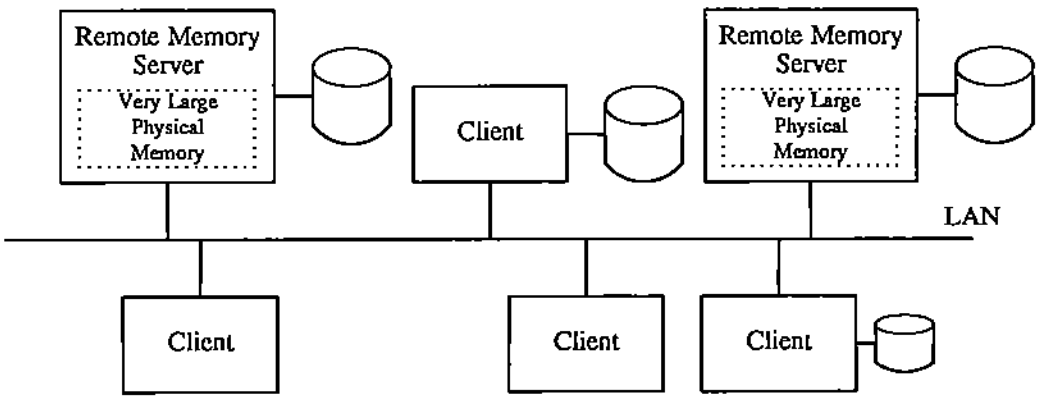


**Figure 1:** An Example Remote Memory Model Architecture

The model uses dedicated, large-memory machines called *remote memory servers* to provide backing storage for virtual memory systems executing on heterogeneous client machines. A virtual memory operating system executes on the client machines and provides the mechanisms needed to efficiently access the large-memory backing storage over a low delay, high bandwidth, communication channel. The large-memory page server machines provide backing storage for multiple heterogeneous client machines at speeds competitive with local disk speeds.

Client machines share the large physical memory resource located at the page server. Each client machine has a private local memory large enough to support the normal processing demands. However, for jobs requiring large amounts of memory, clients page across the network, storing pages on the large-memory page server machine and retrieving them as needed. The page server allocates memory on demand to clients that require additional memory. Unlike other distributed virtual memory systems that page to a remote disk with fixed partitions, the remote memory model provides a large, dynamically allocated, shared, backing storage resource accessible to heterogeneous client architectures.

The goal of the Virtual Memory Xinu project is to design a highly efficient system in which heterogeneous client machines page across the network to shared, memory backing storage. In particular, the project investigates the design of a client virtual memory operating system with efficient support for remote memory backing storage, the design of a highly efficient memory server, and the design of a high performance reliable paging protocol.

# 3. System Design

## 3.1. The Client Operating System

We assume each client machine provides hardware support for virtual memory and network communication required by the operating system to support virtual address spaces and remote memory backing storage. We wanted to design a virtual memory operating system that would work together with a page server to provide fast access to data and yet provide basic operating system functionality such as process management, process synchronization, interprocess communication, and device drivers. More specifically, we identified the following design goals for the client operating system:

**Very Large Programs** Large virtual address space support for programs requiring large amounts of memory.

**Multi-threaded Processes** The ability to execute multiple threads of control within a process, allowing concurrent manipulation of shared data within a process.

**Multi-threaded Kernel** The ability to execute several different tasks concurrently in the operating system kernel.

**Shared Memory** Shared memory primitives that allow sharing of address-dependent data as well as providing an efficient mechanism for communication and synchronization between the threads in a process or between threads in separate processes.

**Hierarchical Design** A design that arranges of the system components into a layered hierarchy resulting in a cleanly and elegantly designed system.

**Remote Paging** Efficient access to a large shared, memory resource for tasks requiring large amount of memory.

**Architecture Independence** The ability to execute the client operating system on the wide variety of architectures composing the heterogeneous environment.

We designed the client operating system with these design goals in mind. We based the design on the Xinu operating system[2,3] because of our familiarity with the system, its simplicity, and its hierarchical design, allowing us to easily modify the system to meet our needs. Because Xinu did not support virtual memory, we were not constrained by an existing virtual memory design. Consequently, we were able to design a virtual memory system that exploits the desirable properties of remote memory backing storage without sacrificing efficiency.

### 3.1.1. Background

Xinu[2,3] is a small, elegant operating system. It arranges the operating system components into a hierarchy of levels, clarifying the interaction between the various components of the system and making the system easier to understand and modify. Despite its small size, Xinu uses powerful primitives to provide the same functionality found in many larger operating systems.

Version 6 Xinu supplied primitives to handle memory management, process management, process coordination/synchronization, interprocess communication, real-time clock management, device drivers, and intermachine communication (a ring network). Version 7 Xinu replaced the point-to-point networking capabilities of Version 6 with support for the Ethernet and TCP/IP Internet protocol software. Version 7 also included a shell and a remote file system that allowed Xinu to access remote files via a remote file server executing on a UNIX system.

Both Version 6 and Version 7 Xinu ran all processes in a single, global, address space and did not use any virtual memory hardware. Before designing the VM Xinu operating system we examined several virtual memory hardware architectures and designed the system to accommodate them. In particular, the design was influenced by two substantially different architectures, the Vax† and the Sun‡ architectures.

### 3.1.2. The VM Xinu Operating System

VM Xinu builds on the functionality provided by the small set of primitives found in Version 7 Xinu. The VM Xinu operating system executes on the set of client machines, providing virtual memory support for tasks requiring large amounts of memory and using remote memory for backing storage.

VM Xinu uses a hierarchical design to incorporate virtual memory support into the operating system, simplifying the kernel and clarifying the relationship between the various components of the system. The design also identifies and separates the architecture dependent components of the system from the architecture independent components. An architecture interface layer resides at the lowest level of the hierarchy, hiding the underlying hardware memory mapping support from the higher layers of the operating system and reducing the effort needed to port the system to new architectures.

VM Xinu uses the hardware's virtual memory support to create a large virtual address space for each application program. Each user process executes in its own address space and can only access data in that space. Each address space is defined by a mapping from virtual memory to physical memory and

† Vax is a registered trademark of Digital Equipment Corporation
‡ Sun is a registered trademark of Sun Microsystems Inc.

uses the MMU support to protect the data in the address space from all other user processes. When starting a user application, the operating system dynamically loads the user's program from the file system into a virtual address space and begins execution of the program. All dynamically loaded user processes execute in user-mode and trap into the kernel via system calls to invoke Xinu kernel routines. Because the kernel is protected from user processes, user processes only access kernel data structures via systems calls.

VM Xinu supports multi-threaded user processes allowing concurrent manipulation of shared data within a process. We define a thread as a point of execution within a process and its associated state information. All threads within a process execute instructions from the same text region, each at a different point in the code, and share the process' data region with all other threads in the process. Semaphores and other interprocess communication primitives provide synchronization between threads, both between threads in the same process and threads in different processes. The system also provides the ability to execute multiple kernel tasks concurrently. Multiple lightweight kernel threads execute at different points in the kernel's text, carrying out kernel operations in parallel. Many operations performed by the kernel such as page reclamation, network management, background paging, etc, are coded very simply and elegantly when viewed as concurrent kernel threads.

Many programming languages provide *address-dependent data structures* to support dynamically allocated data. Sorting and search algorithms often use address-dependent data structures such as linked lists or tree structures that have embedded pointers (virtual addresses). To accommodate sharing of address-dependent data structures, VM Xinu provides two types of data sharing. First, all threads executing within an address space share the data in the address space. Moreover, the data appears at the same location in each thread's view of the address space. Second, VM Xinu allows address-dependent data sharing between threads executing in different address spaces. The operating system reserves part of each address space, called the *shared/private* area. A thread can place address-dependent data in the shared/private area and then allow other threads to access it. Because of a novel design, the data always appears at the same location in all address spaces.

The VM Xinu operating system is unique in that it uses remote (physical) memory, accessed via a high speed, high bandwidth network, for shared backing storage. The kernel provides the virtual memory and networking mechanisms needed to page across the network to a remote memory server. Two kernel *dispatcher* threads handle the multiplexing and demultiplexing of all paging messages sent to and received from the memory server.

The kernel uses a highly efficient, reliable, data streaming, network architecture independent protocol to communicating with the memory server called the *remote memory communication protocol*. The protocol consists of two layers: the Xinu Paging Protocol (XPP) layer and the Negative Acknowledgement Fragmentation Protocol (NAFP) layer. The XPP layer sends and receives high level paging messages while the NAFP layer handles the details of delivering XPP messages over the underlying communication channel. To insure end-to-end[9] reliable storage/retrieval of data, XPP uses timeouts and retransmissions. XPP supports data streaming using asynchronous message delivery, resulting in substantially higher throughput than synchronous protocols. To avoid overrunning the server with paging messages, the XPP protocol provides flow control mechanisms that limit the number of outstanding messages allowed at any given time.

The NAFP protocol fragments XPP messages into one or more packets and then reassembles the message at the destination. NAFP hides the details of the underlying communication channel from the XPP layer and provides transmission of arbitrarily large messages. NAFP attempts to correct fragmentation errors as soon as they occur using negative acknowledgements (NACKs). Early correction of errors using NACKs reduces the number of errors that reach the XPP level and results in improved efficiency with no added overhead in the expected case when no errors occur. We refer the reader to [4] for a more detailed description of the communication protocol.

### 3.2. The Page Server

The page server provides memory backing storage for the virtual memory operating systems executing on the client machines. The design goals we used while designing the page server were:

**Heterogeneous Client Support** — The server should provide remote memory backing storage to heterogeneous clients simultaneously.

**Shared Resources** — Instead of preallocating fixed amounts of memory to each client, the memory resource must be shared between all the clients according to their needs.

**Arbitrarily Large Memory** — The memory resource provided by the server and presented to the clients must appear arbitrarily large.

**Shared Data** — The server should allow clients to share the data stored at the server with other clients.

**Fast Data Access** — To improve overall system performance, the server must efficiently locate and retrieve stored data.

The page server supports heterogeneous client architectures and operating systems regardless of their page size, word size, or byte order. The server uses the transport protocol to transfer pages of any size to or from the heterogeneous client machines, and dynamically allocates data structures to store the variable size memory segments clients send to it. The server does not interpret the stored data; it simply returns the data in exactly the same form in which the data was received.

Instead of preallocating fixed amounts of memory to clients, all clients share the memory resource. When clients request storage space on the server, the server dynamically allocates regions of its memory to clients according to their needs. The only limit on a client is an indirect limit on the collective usage of the resource.

The page server uses a transparent two-level memory scheme to present an arbitrarily large memory resource to client machines. The server connects to one or more disk drives and uses a memory replacement scheme to substantially enlarge the storage capacity of the server. From the client's viewpoint, the server simply provides a single level large memory resource.

To reduce the delay associated with retrieving memory from the remote memory server, the server attempts to minimize the time spent searching the data structures for the desired data. The server uses a double hashing algorithm to locate data in constant time. The data structures used by the server allow clients to share the data they store on the server with other clients. Not only does sharing allow different clients to access the same data, but it also reduces the memory used on the server by eliminating redundant copies of data (common program text, shared libraries, fonts, etc).

## 4. Prototype Implementation

We designed and implemented a prototype distributed system based on the remote memory model. The system consists of heterogeneous client machines (Sun Microsystems SUN 3/50's, Digital Equipment Corporation Microvax I's and II's), a memory server, and a remote file server, all connected by a 10 Mb/sec Ethernet. The Sun and Microvax client machines simultaneously access the remote memory server for backing storage, demonstrating support for heterogeneous clients.

A *page server* provides backing storage for the VM Xinu kernel. Our prototype page server runs as a UNIX user level process, allowing us to run the server on a wide variety of platforms. We have used a SUN 3/50, Sparcstation, Vax 11/780, Microvax II and III, Vaxstation, Decstation, an 8 processor Sequent Symmetry, running a wide variety of operating systems (SunOS, 4.3BSD, Dynix, Ultrix, VM Xinu) as the page server machine. The page server provides backing storage for multiple client machines simultaneously. We built the high speed communication protocol used to reliably transfer pages between the client

and the page server on top of UDP, allowing clients to access page servers on other networks across one or more gateways. The combination of a highly efficient memory server and a special purpose communication protocol results in a high-speed, shared, secondary storage mechanism operating at speeds competitive with a local disk.

## 5. Experimental Results

Initial performance results, obtained from diskless Sun 3/50 client machines paging across a 10 Mb/sec Ethernet to a Sun 3/50 memory server, are described in [4] and show a significant improvement over diskless systems that page to a remote file system (see table 1). The results show that the time to service a page fault in SunOS takes twice as long as the time to service a page fault in VM Xinu. In addition, SunOS pays a high cost to insure that each store operation commits the data to disk before acknowledging the operation. As a result, VM Xinu page store times are 4 times faster than SunOS. In VM Xinu, unlike SunOS, the round trip delay for a write request and the round trip delay for a read request are symmetrical.

| | VM Xinu | SunOS |
|---|---|---|
| **Random Read** | 39 ms | 84 ms |
| **Random Write** | 39 ms | 176 ms |

**Table 1:** Paging access times for VM Xinu and SunOS 4.0 (Taken from [4])

Figure 2 illustrates the memory server response time for various server loads. We used a Sun 3/50 as the memory server with Sun 3/50 machines running VM Xinu as client machines. We generated the various server loads by varying the number of clients and the number of requests per second issued by each client. At 30 requests per second the prototype page server becomes saturated and any additional load on the server significantly increases the round trip delay. However, for loads of less than 30 request per second the results show that the average round trip delay for store and fetch requests remains less than 56 ms, regardless of the number of clients. As long as the combined request rate of all the client machines remains less than the saturation rate, we can add new clients without degrading the performance of existing clients.

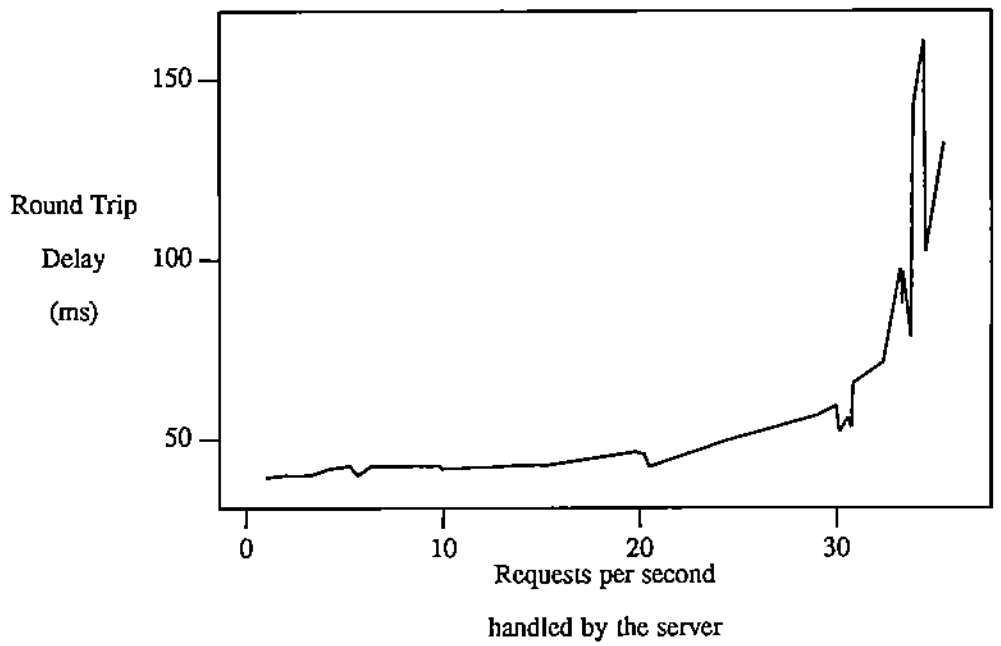


**Figure 2:** Remote memory delay for various server loads

The paging times reported in [4] measure the combined performance of the remote memory server and the communication protocol. Our experience with the system indicated that the communication protocol, rather than the memory server, consumed the majority of the paging time. To improve the paging performance, we focused our attention on the communication protocol in an attempt to streamline the protocol and reduce the communication overhead.

| Component | Time | Percentage |
|---|---|---|
| **XPP/NAFP** | 8.6 ms | 22% |
| **UDP/IP** | 23.6 ms | 61% |
| **Memory Server** | 6.8 ms | 17% |

**Table 2:** Breakdown of the time required to process a paging request.
The percentage is calculated from a total paging time of 39 ms.

Because NAFP hides the underlying communication channel from the XPP protocol, we can run the communication protocol over most network architectures. In our prototype implementation, we built the communication protocol on top of the UDP protocol, using the virtual network provided by the IP protocol to allow access to memory servers on remote networks. Because our design does not bind the

remote memory communication protocol to a specific type of communication channel, we wanted to measure the additional overhead incurred as a result of our implementation decision to use UDP as the communication channel.

Table 2 shows the breakdown of a paging request in terms of the time spent processing each stage of the request. The table shows that the majority of the paging time can be attributed to the UDP/IP protocol (over 60%). Upon closer inspection, we observed that the UDP/IP layer spent 59% (13.9 ms) of its time copying data (i.e. the contents of the page being stored/fetched). Consequently, we optimized the UDP/IP copy routine to efficiently copy the large blocks of data transmitted in NAFP messages. The optimization reduced the copy time from 13.9 ms to 4.0 ms and reduced the paging times from 39 ms to 32 ms. Even with the optimized copy routine, the UDP/IP layer consumes 42% of the paging time. Consequently, building the communication protocol directly on the communication channel rather than on UDP would substantially improve paging performance.

| | Sun 3/50 (SunOS) | Sun 3/50 (VM Xinu) | Vaxstation 3100 (Ultrix) | Sparc 1+ (SunOS) | Decstation 3100 (Ultrix) |
|---|---|---|---|---|---|
| **Store** | 32 ms | 31 ms | 30 ms | 22 ms | 23 ms |
| **Fetch** | 32 ms | 32 ms | 34 ms | 25 ms | 29 ms |

**Table 3:** Paging times for several different page server architectures.
The time shown is the round trip delay to store/fetch an 8K byte page.

Table 3 shows paging times for a Sun 3/50 client paging across a 10 Mb/sec Ethernet to a wide variety of architectures. We executed the memory server as a user level application on each architecture and measured its performance. Each column of the table shows the time required to store/fetch an 8K byte page to/from the specified architecture/operating system. The times show that the architecture and the operating system both have a significant impact on the paging times. Clearly, the RISC architectures (Decstation 3100 and Sparc 1+) outperform the older CISC architectures (Sun 3 and Vaxstation). In particular, the RISC machines provide backing storage at speeds competitive with local disks. Moreover, because our implementation build on the UDP/IP protocols, the operating system's implementation of these protocols has a significant effect on paging times. SunOS sends and receives packets at roughly the same speed, while Ultrix sends packets substantially slower than it receives packets. Although the optimized copy routine reduces the time VM Xinu spends processing a message by 9.9 ms, we only see a 7 ms speed-up over the old Sun 3/50 page server times (table 1). In this case, SunOS is the bottleneck. The

lower bound on paging times (32 ms) is based on the rate at which SunOS can process UDP/IP packets.

In short, the results show that our implementation decision to use UDP/IP as the underlying communication channel significantly impacts the system's performance. However, even with the extra overhead resulting from UDP/IP, the system performs at speeds competitive with a local disk. The prototype clearly demonstrates the viability of building hierarchically-layered systems paging to remote memory backing storage without sacrificing efficiency.

## 6. Conclusions

Using the remote memory model as an alternative model for designing distributed systems has many attractive properties. The large memory resource shared by all client machines is especially appealing. Experience with the prototype system clearly demonstrates the viability of the remote memory model and shows that performance is competitive with distributed systems currently in use. Finally, we showed that the remote memory model can support heterogeneous clients machines without sacrificing efficiency.

### References


1. Maurice J. Bach, *The Design Of The Unix Operating System*, Prentice Hall, 1986.
2. Douglas Comer, *Operating System Design: The Xinu Approach*, Prentice-Hall, 1984.
3. Douglas Comer, *Operating System Design, Volume II: Internetworking with Xinu*, Prentice-Hall, 1987.
4. Douglas Comer and James Griffioen, "A New Design for Distributed Systems: The Remote Memory Model," Proceeding of the Summer Usenix Conference, June 1990.
5. Samuel J. Leffler, Marshal K. McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison Wesley, 1989.
6. Henry Levy and Peter Lipman, "Virtual Memory Management in the VAX/VMS Operating System," *Computer*, pp. 35-41, March 1982.
7. Michael N. Nelson, "Virtual Memory for the Sprite Operating System," Tech Report UCB/CSD 83/301n, University of California Berkeley, June 1986.
8. James L. Peterson and Abraham Silberschatz, *Operating System Concepts*, Addison Wesley, 1985.
9. J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, pp. 277-288, 1984.
10. Robert A. Gingell and Joseph P. Moran and William A. Shannon, *Virtual Memory Architecture in SunOS*, Sun Microsystems, Inc., 1988.
11. Brent B. Welch, "The Sprite Remote Procedure Call System," Tech Report UCB/CSD 86/302, University of California Berkeley, June 1986.