1989

A Virtual Memory Operating System for a Distributed Workstation Environment

James Griffioen

Report Number:
89-884

A VIRTUAL MEMORY OPERATING SYSTEM FOR A DISTRIBUTED WORKSTATION ENVIRONMENT

James Griffioen

CSD-TR-884
May 1989

Thesis Proposal

A Virtual Memory Operating System For A Distributed Workstation Environment

James Griffioen

May 22, 1989

Abstract

Recently we have seen an explosive growth in the number of high performance, low cost workstations available on the market. An increase in the usage and popularity of new applications like windowing systems and object oriented applications has resulted from this growth. These applications rely heavily on data sharing. To support these applications, the operating system must support efficient shared memory and message passing. This paper proposes an efficient shared memory mechanism that also allows sharing of *location dependent data* such as linked lists and tree structures. In addition, the paper proposes efficient support for *persistent objects* used in object oriented applications. Finally, the paper proposes an efficient mechanism that takes advantage of diskless workstations and high speed LAN's to page remotely to a dedicated *page server* machine with performance similar to systems paging to a local disk.

1 Introduction

Recently we have seen an explosive growth in the number of high performance, low cost, workstations available on the market. Processing power, as well as network speeds, continue to increase, while the cost of memory continues to decrease. As a result, new applications such as windowing systems and object oriented applications have been increasing in usage and popularity.

Windowing systems and object oriented applications frequently involve a considerable amount of message passing and data sharing. An efficient shared mem-

ory mechanism is a useful and necessary component in such a system. Interprocess communication can be implemented efficiently in a shared memory system. In addition, many conventional programming languages support pointers and data structures with embedded addresses. The shared memory mechanism should be able to efficiently support sharing of *location dependent data structures* such as linked lists and tree structures.

Many recent virtual memory systems have implemented shared memory [Ras86][GMS88][Bac86], however, some pay a high overhead for shared memory and most have, at best, a limited form of location dependent data sharing. The Mach operating system[Tev87] allows processes to read a region from another process's address space into its own address space, but actually receives a copy of the region. It may then write into the other process's address space by copying the entire region into the other process's address space. Because of the copying, the pages are not physically shared, decreasing efficiency and leaving coherency problems for the user to deal with. Futhermore, although the user may request that the shared region be mapped to the same location in both address spaces, there is no guarantee that it will be, in which case, the system will place it at a location it finds suitable. System V Unix [Bac86] and SunOS[GMS88] suffer from the same problem and are not able to guarantee that the region will be placed at the same address in all the processes desiring to share the memory. Both Mach and Sprite[Nel86] allow sharing of location dependend data in the heap area, but only among a parent and its children.

Persistent objects are memory objects (code and/or data) that continue to exist (persist) after the last active computation on the object has completed. Persistent objects are an important part of most object oriented systems [DJA88] [ABLN85]. Such objects do not map nicely onto processes in most conventional virtual memory systems because processes are terminated when the computation has ended. In addition, it has been shown that users tend to use a small set of programs repeatedly[Kri87]. This usage pattern indicates that performance

can be improved if the user's small set of programs reside in *soft-lock* persistent objects. Soft-lock persistent objects are persistent objects which have a guaranteed minimum lifetime in memory after the last active computation on the object has completed. If the object has not been reactivated before the end of the minimum guaranteed lifetime, the object may be removed from the system. Soft-lock persistent objects would allow frequently used programs and data to remain in virtual memory so they can be reactivated more quickly and efficiently than infrequently used programs.

Both Clouds and Eden are specially designed systems with support for persistent objects and object reactivation[DJA88][ABLN85]. However, objects must be explicity removed, and there is no notion of soft-lock objects. Some implementations of Unix use a *sticky bit* which allows programs to remain in the swap partion after they have terminated so they can be restarted quickly[Bac86]. These systems lack mechanisms for creating persistent objects, and they only allow the code segments to persist (data segments are lost).

Recent changes in technology, together with the economics involved, have made paging to remote storage, rather than a local disk, a practical and economically desirable goal. Increasing CPU speeds and network speeds, coupled with decreasing costs of memory and powerful workstations, have made this possible.

Hierarchical operating systems [Com84] collect each policy (page replacement policy, memory allocation policy, scheduling policy, etc) together into one place making the system easier to understand. Experience with the simplicity and flexibility of hierarchical operating systems convince us that it is desirable to try to design virtual memory systems in a hierarchical fashion.

1.1 Thesis

Our thesis is that it is possible to design and implement a hierarchical virtual memory operating system with efficient support for shared memory and soft-lock persistent objects. In addition, we argue that it is possible to design a system

that pages to remote storage with performance similar to or better than systems that page to local disk. We propose to study the system design issues, algorithms, and network protocols needed to build such a system.

1.2 Project Goals

The goals for the project are:

Shared Memory We want a shared memory mechanism that is efficient, making shared memory access as efficient as non-shared memory access. The shared memory mechanism should provide limited (as opposed to global) sharing; protecting the memory being shared from processes without the right to access it. In addition, we want to be able to share location dependent data (data with embedded addresses).

Persistent Objects We would like to have efficient support for objects and, in particular, soft-lock persistent objects. This includes methods for fast reactivation of persistent objects.

Remote Paging We would like to use diskless workstations paging to remote storage and still maintain performance similar to workstations with a local disk. We hope to reduce the cost of the system (economic cost) and yet achieve similar or better performance.

Elegant Design The system should be designed in a fashion that is easy to understand and modify.

1.3 Research Issues

To accomplish the goals listed above, we propose to look at the following research issues:

- We propose to investigate a shared memory mechanism that will support location dependent data sharing. We also plan on studying the shared

memory operations/interface that the virtual memory system will present to the user.

- We would like to study methods for separating the lifetime of a process from the activity of a process. Furthermore, we would like to investigate methods for fast reactivation of persistent objects.

- Paging over the network to remote storage with performance similar to systems with a local disk requires efficient protocols. We plan to study specialized communication protocols that can yield high throughput and low delay for remote paging. In order to make the pager server as efficient as possible, we propose to investigate efficient page lookup and storage alogithms for the page server.

- Experience with hierarchical operating systems shows that a hierarchical design makes the system easier to understand and modify[Com84]. We would like to determine if virtual memory management can be incorporated into a hierarchical design.

The overall system architecture we envision is pictured in Figure 1. The system consists of numerous diskless *client* machines, a *page server* machine, a *file server* machine, and a number of miscellaneous server machines. The client machines are completely independent of each other, yet they all page to the same page server.

2 Proposed Research

2.1 Integrating VM management into the Hierarchical Design

We propose to carefully design the virtual memory management to fit into the hierarchical design already present in Xinu. We would like to bring all the virtual memory policies together into one place. In Version 7 Xinu (non-virtual memory),

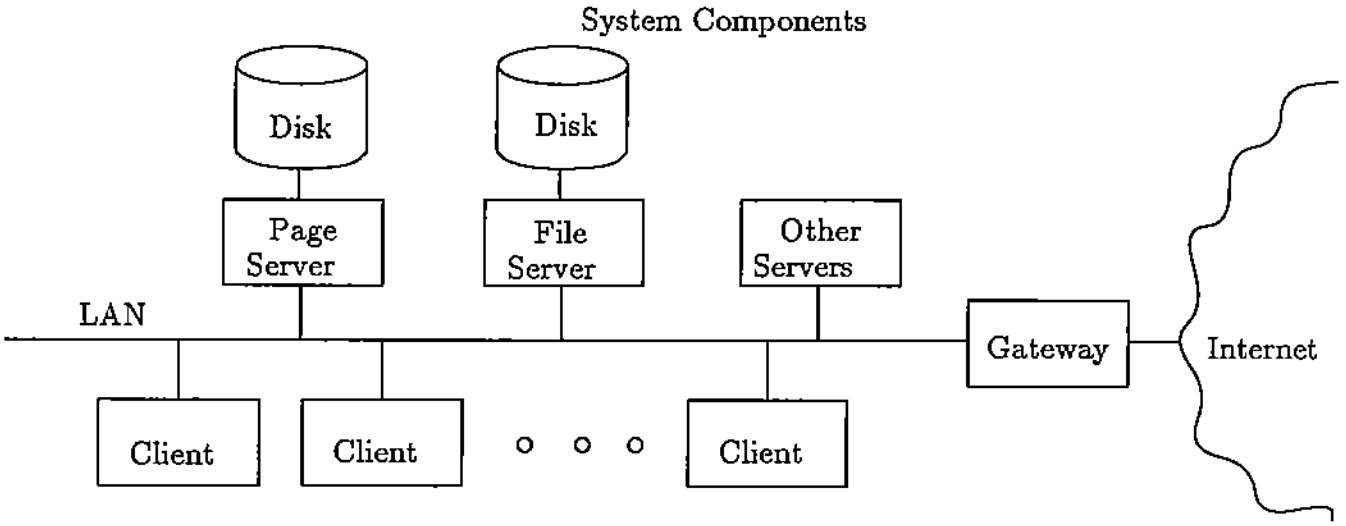

Figure 1: System Architecture: Many clients, a page server, a file server, and other servers such as a RARP server, timeserver, and nameserver

the memory management routines were divided into high and low level memory management routines. Similarly, we believe that the physical and virtual memory management routines in a virtual memory system can be divided into high and low level routines/processes.

We feel the physical memory management routines can be broken into low level routines used to map (and unmap) virtual pages to physical pages while the high level routines will implement the page replacement algorithms. Similarly, the virtual memory management routines can be split into low level routines used to allocate stack space and kernel memory while the high level routines manage the shared memory and user's heap.

We would also like to incorporate an *architecture interface layer* into the hierarchy to reduce the amount of effort needed to port the system to a different architecture. The architecture interface layer will present generic page table entries and methods for accessing them to all upper layers. Upper layers will then be independent of the virtual memory mechanisms supported by the hardware.

2.2 Address Spaces and Threads

The notion of a process in conventional time-sharing systems is defined by an address space and a point of execution within that address space. When the computation has been completed, the process (both the point of execution and the address space) is removed from the system. In order to separate the lifetime of a process from the activity within a process, we propose to break the conventional process into multiple points of execution called *threads* which execute in memory objects called *address spaces.*

An *address space* is simply the virtual memory used to store the data and the code associated with a particular job. A *thread* is a point of execution within an address space. Threads are defined by a program counter, stack pointer, and stack memory. An address space may be inhabited by 0 or more threads. This means that address spaces are independent of threads and may continue to exist even after the last thread is done. This allows frequently used programs to remain in memory for fast reactivation. In addition, threads in the same address space share all the data in that address space.

2.3 Shared Memory

To insure that location dependent data can be shared between threads in different address spaces as well as with threads in the same address space, we propose to reserve a portion of the address space for sharing called the *shared/private region.*

The shared/private region is a reserved portion of every address space. The region begins and ends at the same fixed locations in every address space (the start and end are set by the kernel). To share data with a thread in another address space, a thread makes a system call requesting memory from the shared/private area. The thread then makes a system call requesting that the memory it just received be mapped into the address space of the thread that needs to share the data. Both threads share the same physical pages. The pages are mapped to the same virtual address in both thread's address spaces so location dependent data

sharing is possible.

Since the shared/private area is part of an address space, it is accessible to all the threads in an address space. Because threads give away access, the shared/private region provides an efficient capability based sharing mechanism with capabilities given to address spaces, not individual threads.

To improve system call times, we also propose to reserve a portion of each threads kernel stack for passing data to/from the kernel called the reserved shared area (RSA). The RSA is guaranteed to be mapped to physical memory at all times, allowing the kernel to access the RSA without the fear of page faulting. System calls are speeded up because there is no need to copy data between the user area and the kernel area.

2.4 Paging To Remote Storage

In order for a diskless client paging over the network to remote storage to achieve performance similar to or better than a system paging to a local disk requires an efficient protocol and page server. To achieve this goal we propose to investigate a specialized reliable *paging protocol* with very low overhead and low delay, and we propose to investigate page look-up and storage algorithms for the page server.

The paging protocol we are investigating is simple, using only four types of messages: page read requests, page write requests, address space/thread create messages, and address space/thread temination messages. It assumes the underlying protocol is unreliable, so it uses acknowledgements and timeouts with retransmissions to guarantee reliability.

Because that paging protocol guarantees reliability, it can be built on any datagram protocol such as UDP[Pos80b] or VMTP[Che86] (assuming fragmentation is done at some level). However, because the page size on many conventinal computers is larger than the maximum network packet size, store/fetch request may be broken up into many packet fragments. For example, to send or receive an 8K SUN3/50 page over an ethernet requires a minimum of 6 (1536 byte) packets.

UDP/IP fragments are not acknowledged[Pos80a]. If any one of the 6 packets is lost, all six must be retransmitted by the paging protocol. VMTP allows selective acknowledgement of fragments; however, every message is acknowledged by VMTP which increases the overhead and reduces the throughput[Che88]. Our desire is to improve reliability without degrading throughput.

The paging protocol will be built on a specialize transport protocol that does fragmentation and improves reliability using negative acknowledgements. Each part of a fragmented page is assigned a sequence number and sent in sequence. As packets arrive on the receiving end, they are reassembled but are not acknowledged. The receiving end remembers the sequence number of the last fragment for each page that is currently arriving. As soon as a fragment comes in out of sequence, a negative acknowledgement containing the sequence number of the missing fragment is sent to the transport layer of the sender. The sender then backs up and begins sending from the missing fragment again. This protocol is extremely efficient and optimizes for the expected case; the case where very few packets are lost.

The performance of the entire system hinges a great deal on the performance of the page server. To achieve maximum performance, we propose to investigate a *page access algorithm* that stores and retrieves pages in almost constant time.

Each store/fetch request the page server receives is uniquely identified by the machine id, address space or thread id, and page number of the desired page. By applying a double hashing function to this information, the page server can locate the page in almost constant time[Knu73]. Even if the hash table is 95% full, the average number of probes needed to locate a page will be no more than three.

The page server will also use a *two-level backing store* to minimize access time. The first level of the backing store is the memory of the page server and the second level is the disk(s) connected to the page server. The server caches as many pages as possible in memory for fast access and writes all remaining pages

to disk. By placing additional memory in the page server machine, clients are able to run jobs requiring a large amount of memory without having that much memory locally.

Finally, we would like to investigate data structures and algorithms that will allow the page server to serve heterogeneous clients simultaneously.

3 Research

3.1 Research completed

We currently have a prototype implementation running on a Digital MicroVAX-I and a Sun Microsystems SUN3/50. The system pages to a remote page server running on a Digital VAX11/780. The page server is able to serve both architectures simultaneously.

The prototype is designed in a hierarchical fashion with an architecture interface layer that makes all upper level routines machine independent. Address spaces and threads as described earlier are supported. Address space may have 0 or more threads of control running in them at any given time. The shared memory region is reserved in every address space and mechanisms for mapping a page into multiple address address space are in place. However, because the shared memory allocation/sharing primitives have not been designed, the region cannot be used. The RSA area is also implemented and is used by dynamically loaded processes in the UVAX-I version. Virtual memory management is also in place and uses a global page replacement policy. The remote paging mechanism uses the paging protocol to store/fetch pages from the page server.

Our preliminary results indicate that the average page store/fetch times between the UVAX I and the VAX11/780 are on the order of 40-50ms. This seems reasonable since ICMP echo requests, which are small packets handled at the kernel level, take 23ms between the UVAX I and the VAX11/780. Paging requests are usually much large packets than ICMP echo packets and are handled by a user level page server process which may have to go to disk for the needed

data. We are just beginning to gather results from the SUN3/50 (a much faster machine than the UVAX I) that appear to be much better than the UVAX I results.

3.2 Proposed Research

I propose to continue researching the design of *address spaces/threads* and *shared memory*. I plan to develop efficient shared memory primitives for allocating, dellocating, protecting and sharing memory, and also the kernel data structures and algorithms need to support these primitives. I then plan to implement them in the prototype and compare them to existing shared memory systems to show the efficiency and functionality differences. I also plan on investigating methods for managing soft-lock persistent objects. The issues involved are the kernel data structures used as well as the methods for creating/removing/reactivating persistent objects. I then hope to show the efficiency advantages of using the soft-lock mechanism over the conventional approach of immediately terminated objects or permanent objects.

4 Summary

We propose to study the system design issues, algorithms, and network protocols needed to build a virtual memory system with efficient support for shared memory, persistent objects, and remote paging.

The main contribution of our work will be an increased understanding of the issues involved in designing efficient mechanisms that support shared memory. In particular, it will result in an increased knowledge of the issues involved in location dependent data sharing. Our work's second contribution will be an understanding of how soft-lock persistent objects can be integrated into the hierarchical design of a virtual memory system. A third contribution of this work will be to develop a remote paging system with better performance than current remote paging systems.

We expect the thesis work will be completed when the following steps are finished:

- We complete the design of a virtual memory system with mechanisms for efficient shared memory, persistent objects, remote paging, and virtual memory management.
- We implement a prototype based on the design and test its viability and efficiency.
- We compare our shared memory to other existing shared memory systems to show the functionality advantages and the efficiency advantages or our system.

5 Acknowledgement

We wish to thank Guoben Li for his suggestions and all the effort he devoted to the implementation of the prototype.

References

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazawska, and Jerre D. Noe. The Eden System: A Techincal Review. *IEEE Transactions on Software Engineering*, SE-11:43–58, January 1985.

[Bac86] Maurice J. Bach. *The Design Of The Unix Operating System.* Prentice Hall, 1986.

[Che86] David R. Cheriton. VMTP: a transport protocol for the next generation of communication systems. In *SIGCOMM '86 Symposium*, pages 406–415, ACM, August 1986.

[Che88] David Cheriton. Vmtp: versatile message transaction protocol. ARPANET Working Group Requests For Comments, Febuary 1988. RFC 1045.

[Com84] Douglas Comer. *Operating System Design the XINU Approach.* Prentice–Hall, 1984.

[DJA88] Parha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The Clouds Distributed Operating System. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9, IEEE, June 1988.

[GMS88] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual Memory Architecture in SunOS. 1988.

[Knu73] Donald E. Knuth. *Sorting and Searching.* Addison Wesley Publishing Company, 1973.

[Kri87] Balachander Krishnamurthy. *A Uniform Model of Interaction In Interactive Systems.* PhD thesis, Purdue University, West Lafayette, Indiana, December 1987.

[Nel86] Michael N. Nelson. *Virtual Memory for the Sprite Operating System.* Technical Report UCB/CSD 83/301n, University of California Berkeley, June 1986.

[Pos80a] J. Postel. DOD Standard Internet Protocol. January 1980. RFC 760.

[Pos80b] J. Postel. User Datagram Protocol. August 1980. RFC 768.

[Ras86] Rick Rashid. Threads Of A New System. *Unix Review*, 4:37–49, August 1986.

[Tev87] Avadis Tevanian. *Architecture Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* Technical Report CMU-CS-88-106n, CMU, December 1987. Ph.d. Thesis.