1989

Virtual memory for Diskless Desktop Workstations

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

James Griffioen

Report Number:
89-916
Virtual Memory for Diskless Desktop Workstations

Douglas Comer
Computer Science Department
Purdue University
West Lafayette, IN 47907
(317) 494-6009
comer@cs.purdue.edu

James Griffioen
Computer Science Department
Purdue University
West Lafayette, IN 47907
(317) 494-7836
jng@cs.purdue.edu

September 29, 1989

Abstract

New applications, such as windowing systems, WYSIWYG (what you see is what you get) editors, and object-oriented applications are increasing in popularity and use. The popularity of such applications is sparked, in part, by the recent explosion in sales and availability of low-cost, high-performance workstations. Many of the new applications require large amounts of memory, making virtual memory a crucial component of a workstation operating system. Virtual memory mechanisms allow each application to operate as if it has a very large amount of memory, even though the amount of physical memory is much smaller.

Virtual memory has been available on mainframes and minicomputers for the past decade. However, virtual memory support on workstations differs from that of large machines because workstations are designed to operate in a distributed networked environment. Workstations use hardware and software specifically designed to support network access. Workstations can use remote computers/devices for virtual memory backup instead of using a local disk.

This paper describes the methods and mechanisms underlying virtual memory. It begins with a brief description of the hardware support for virtual memory provided by current workstation architectures, and then goes on to show how the operating system uses that hardware support. It describes how the operating system moves portions of memory to secondary storage and back, to create the illusion that each application has much more memory available than is physically present on the machine.

Finally, the paper shows how the network, together with a remote storage device, can be used to replace local disks as the secondary storage medium without degrading system performance. It presents a novel new paging protocol that demonstrates one possible method for designing workstation memory systems. The scheme, which uses a local area network for remote paging, has low overhead and uses a special-purpose high-speed protocol to achieve high throughput and low delay. The result is a virtual memory system that pages remotely with performance comparable to systems that page to a local disk.

Keywords: virtual memory, shared memory, paging, page replacement, remote paging, network
1 Introduction

Rapid changes in chip technology, cpu speeds, and memory sizes have produced dramatic changes in the applications and use of conventional computers. These technological changes have also caused an explosive growth in the number of high performance, low cost workstations available on the market. Workstations are capable of performing tasks that were only possible on mainframes and minicomputers of the past. The increasing popularity of applications such as windowing systems, WYSIWYG (what you see is what you get) editors, typesetting, elaborate word processing with grammar and dictionary aids, and object oriented applications is due, in part, to the availability of bitmapped displays and low cost computing power provided by workstations.

Current workstations perform applications that personal computers of the past could not perform because current workstations have larger memories, faster CPU's, networking support, and most importantly, virtual memory hardware. The applications mentioned earlier (windowing systems, WYSIWIG editors, etc.) require large amounts of memory, often much more memory than is normally available on personal computers. Even with the larger memories available on conventional workstations, a single application can consume all (or at least a substantial portion) of the physical memory. Workstations, as opposed to personal computers of the past, can execute applications requiring large amount of memory because they have virtual memory hardware support. It is the virtual memory hardware support that makes large applications possible. Virtual memory hardware allows each application to operate as if the machine has a large amount of memory, even though the actual amount of physical memory is much smaller.

The virtual memory hardware also allows applications to share data and pass messages efficiently by sharing the physical memory. Allowing shared physical memory is crucial to object oriented applications and windowing systems that rely heavily on

\[1\] Memory sizes of 512K bytes to 1M bytes are typical for personal computers

\[2\] Workstations often have 4 to 16 Mbytes of memory
data sharing and message passing.

The physical memory in a workstation is called the primary storage. Secondary storage is a device or machine that stores the portions of the address space that cannot fit into the workstations memory. Magnetic disks have been the main form of secondary storage, providing the speed and bandwidth needed to make virtual memory possible. Workstations are intentionally designed to operate in a networked environment, allowing for new secondary storage methods. Workstation hardware supports high speed networking, making it possible for diskless workstations to use remote machines or devices for secondary storage. Portions of the address space that do not fit in the workstation’s primary memory are sent across the network to a remote machine (or device) where they are stored until they are needed. High speed local area networks together with fast CPU’s and large physical memories provide the speed and capacity needed to support remote secondary storage resulting in system performance similar to, or better than, systems that use a local disk for secondary storage.

In the remainder of this paper we will describe the hardware and software found in conventional workstations needed to support the new applications mentioned earlier as well as old applications, previously only possible on mainframe computers. We begin by reviewing the hardware support provided by conventional workstations. We then describe how the operating system uses the hardware to create the virtual memory illusion. Finally, we present a novel new paging protocol that allows a diskless workstation to page to remote storage and yet attain performance similar to systems that page to a local disk.

2 Workstation Hardware Support

In the past, only mainframes and minicomputers had hardware support for virtual memory and networking. Personal computers, which had relatively small memories\(^3\), had no virtual memory support and rarely any network support. Conventional work-\(^3\)

\(^3\)1M byte or less
stations have hardware support for virtual memory similar to the hardware virtual memory support found on mainframe machines. In addition, workstations support high speed networking allowing them to communicate and share resources with other workstations.

2.1 Physical and Virtual Memory

Virtual memory hardware support allows applications to believe that the memory space is much larger than the physical memory installed on the machine. The hardware presents a virtual address space to each application. Virtual address spaces are often as much as 1000 times larger than the size of the physical memory. Usually the size of the virtual address space is based on the size of the BUS. Many workstations have processors that can handle 32 bit addresses, resulting in a potential virtual address space of 4 gigabytes (2\(^{32}\) bytes). However, the physical memory on the workstation may only be between 4 and 16 megabytes.

At any given time, only a small portion of a virtual memory space is stored in physical memory. Programs execute using virtual addresses to refer to memory locations. The hardware translates every virtual address used by the program into a physical address. Because virtual to physical address mappings change continually (depending on what part of the program needs to be in memory), the hardware uses translation tables to convert virtual addresses to physical addresses. The operating system creates and manages the translation tables used by the hardware.

Both physical memory and virtual memory are divided up into pages. A page is simply a fixed length section of contiguous memory locations, where the fixed length is usually a power of 2 (called the page size). Memories used to be small and expensive, so page sizes tended to be small (typically around 2\(^9\) = 512 bytes) to maximize memory utilization. Current workstations have much larger memories, in part, because the cost of memory has dropped significantly and the density of memory on a chip has increased. Because workstations have much more memory, they can use

---

\(^4\)Local Area Networks with speeds of 10 Mbits/sec are common
larger page sizes\(^8\) to reduce the page table size and improve paging performance. Page size can dramatically effect the performance of an application program. In section 4.4 we demonstrate the effect page size can have on various types of applications.

The translation tables used by the hardware contain one entry per virtual page. Each entry tells the hardware how to map the virtual page to the corresponding physical page. These translation tables are usually called \textit{page tables}.

### 2.2 Page Tables

A page table specifies the mapping between a virtual address space and the physical space. There is one \textit{page table entry} for every page in a virtual address space. A page table entry is a structure consisting of multiple fields, which, when interpreted all together, tell how to map a virtual memory location to a physical memory location (or to secondary storage). Some of the values in the page table entry are set and used by hardware, and some of the values are reserved for the software (usually the operating system) to use.

The specific values stored in a page table entry differ significantly from architecture to architecture. However, a few page table entry values are common to most architectures: a \textit{page number}, a \textit{hardware valid} bit, \textit{protection} bits, a \textit{modified} bit, and a \textit{referenced} bit. The \textit{page number} contains the physical page number where the virtual page can be found. When the operating system moves a virtual page from physical memory to secondary storage, it stores the disk address where the page can be found in the page number field of the page table entry. The \textit{hardware valid} bit indicates whether the virtual page can be found in physical memory or not. If the hardware valid bit is not set, the page is stored on secondary storage or is uninitialized. If the hardware valid bit is set, the page is in physical memory. The virtual memory hardware then checks the \textit{protection} bits to make sure that the page can be referenced in the current CPU mode (either \textit{privileged} or \textit{user} mode). The virtual memory hardware then combines the page number and the offset within the page to

\(^8\) 4 - 8 Kbytes
determine the complete physical address.

Figure 1: Hardware page table, mapping virtual to physical memory

When a page is modified (written to), the hardware sets the modified bit, and when a page is referenced (written to or read from), the hardware sets the referenced bit. The modified bit indicates whether or not the page has changed since it was last read in from secondary storage. If the operating system wants to replace a page that has been modified, it must first write the page to disk. If the page has not been modified, then the copy on secondary storage is still valid and the page does not need
to be written out. The modified and referenced bits are also used by software to determine which physical page to replace when there are no more free pages available (see figure 1).

2.3 Context Registers

So far we have discussed how the hardware maps a virtual address space onto physical memory. The hardware is actually capable of mapping multiple virtual address spaces onto physical memory. Workstations, unlike personal computers, are designed to handle multiple tasks simultaneously. For this reason, workstation hardware is designed to support multiple virtual address spaces. The operating system assigns one virtual address space to each application program (process). The hardware expects each virtual address space to have its own page table.

The hardware context register specifies which page table the hardware should use when mapping virtual addresses to physical addresses. When the operating system is suspending the execution of process A in order to switch to process B, it changes the context register so the page table associated with process B will be used. Although the actual implementation differs considerably from workstation to workstation, there is, in general, a register that indicates which page table to use.

Given the context register and the page table specified by the context register, the hardware is able to translate a virtual address to a physical address. Assuming the desired virtual memory location is currently in physical memory, the hardware translates a virtual address to a physical address using the following algorithm:

1. Consult the context register to locate the current page table.

2. Calculate the virtual page number from the virtual address.

3. Find the correct page table entry based on the virtual page number.

4. After checking the hardware valid and protection bits, use the physical page number stored in the page table entry to locate the physical page.
5. Calculate the offset within page from the virtual address.

6. Use the offset within page and the physical page number to calculate the physical address of the memory location.

Every virtual address goes through this translation process. Because programs tend to access memory locations that are relatively close together (this phenomena is know as the principle of locality), the hardware frequently ends up accessing the same page table entry repeatedly to find the physical page number. To make the translation process more efficient, most workstation architectures use a translation look-aside buffer. A translation look-aside buffer is simply a cache of the most frequently used page table entries. When a page table entry is used, it is placed in the translation look-aside buffer so that it will not have to be fetched from memory the next time it is needed. When the operating system switches from one process to another, it must clear the translation look-aside buffer so that the page table entries associated with the new process will be used.

2.4 Page Faults

In the preceding section, the address translation method assumed that the desired virtual memory location was in physical memory. If this is not the case, then translation proceeds as normal until step 4 when the hardware valid bit is checked. If the page is not physically in memory, the hardware valid bit will not be set, and the hardware will signal a page fault. A page fault is a hardware interrupt and is handled the same as all other hardware interrupts. A value, indicating the reason for the page fault, is pushed onto the stack and a page fault handler routine (part of the operating system) is invoked to handle the page fault. In most cases the page is on secondary storage. Once the page has been brought in, the page fault interrupt handler returns, and the hardware begins the translation process for the virtual address all over again. A page fault may occur because the hardware valid bit is not set, or because the protection bits do not allow access to the page.
2.5 Hardware Networking Support

Many conventional workstations come equipped with high speed network hardware. The interface provided by the networking hardware allows the software to send and receive packets, and to query the interface for configuration information. Many network hardware interfaces have a network address assigned to the network hardware. The software can query the network hardware to obtain this address, which it may then send to other machines so that they will know where to send reply packets. Depending on the network architecture, the hardware may also allow packets to be broadcast to all machines on the network, and it may even allow multicasting; a way to broadcast to a specific group of machines.

3 Paging: Creating the Large Memory Illusion

The operating system uses the virtual memory support provided by hardware to create the virtual memory illusion. The idea is that only a small portion of an application needs to be resident in physical memory at any given time. Various methods such as overlays, swapping, and demand paging have been used to fabricate the large memory illusion.

Overlays are the portions of the application’s memory space that are never used simultaneously. Knowledge about the structure of the program, the code, and its data structures is required to determine what parts can be overlayed. Assuming the overlays are known, the application can be written in such a way that only one overlay is physically present in memory at any given time. The application is responsible for knowing when an overlay is needed, writing out the current overlay, and reading in the needed overlay. Although use of overlays requires a considerable amount of knowledge about the application and the underlying system, overlays have the advantage that they require no hardware or operating system support.

Swapping is a method used by the operating system to temporarily remove an entire application from physical memory. If an application is going to be blocked for
a long period of time (for example, waiting for I/O), large amounts of memory can be freed up quickly by removing the entire application from physical memory.

Demand paging is a method used by the operating system to temporarily remove individual pages of an application. Under demand paging, the operating system attempts to retain the pages an application will need in memory. If a page is not in memory, a hardware page fault is signalled and the operating system brings in the missing page (on demand).

3.1 Paging Data Structures

The operating system manages and allocates all physical memory. It uses several data structures to help manage and keep track of how the physical memory is being used. We have already seen how hardware uses page tables to map virtual addresses to physical addresses. Page tables are used by both hardware and software. The operating system creates and initializes the page tables for the hardware to use. Some of the bits in the page table are set and used by the hardware, but usually there are some unused page table entry bits that can be used by the operating system. The operating system uses some of the bits to record where the page is if it is not in
physical memory, or it uses the bits to indicate that a virtual page is not initialized.

A second data structure used by the operating system is the frame table. A frame is another name for a physical page. A frame table is simply an array of records, with one record per physical page (frame) of memory. Each entry in the array maintains information about the physical page it represents. An entry may indicate what process is using the physical page, whether or not the page is locked in memory (a locked page cannot be removed from physical memory), whether or not the page is currently being paged in or paged out, and a pointer back to the page table entry of the virtual page (see figure 2).

In addition to a frame table, the operating system uses various lists to keep track of the frames that are available and the frames that are in use. The number of lists used and the mechanisms for managing the lists vary from operating system to operating system, but three frequently used lists are the free list, the active list, and the modified list. The free list is a list of all the frames that are not currently being used. The active list is a list of the frames currently being used, and the modified
list is a list of all the frames that should be reclaimed but must be written out to secondary storage first because they were modified. Thus, one additional piece of information each frame table entry contains is a list pointer that links it into one of these three lists. When the operating system needs a frame to use, it takes it from the free list. When the operating system decides a frame can be replaced, the operating system moves the frame from the active list to the free list if the frame has not been modified, or from the active list to the modified list if the frame has been modified.

### 3.2 Handling Page Faults

We mentioned earlier that the hardware raises a *page fault* interrupt when it cannot complete a virtual to physical address translation. There are several reasons why the translation process may fail. It is the job of the operating system to locate and correct the problem. When a page fault interrupt occurs, the interrupt dispatcher invokes the *page fault handler routine* to service the interrupt. The page fault handler routine determines the cause of the page fault and takes the appropriate action.

The most common type of page fault occurs when the virtual page being referenced is not in physical memory. In this situation, the hardware valid bit in the page table entry is not set. The page fault handler must then determine why the desired virtual address is not in physical memory. In general, there are four possible reasons why a virtual page is not in physical memory. First, the program may have referenced an invalid memory location; a virtual memory location out of the region allocated to the process. The page fault handler either removes the faulting process or invokes an error handler to recover from the error. Second, the page may have been paged out to secondary storage. The page fault handler obtains a free frame from the free frame list, fetches the page from secondary storage, and changes the page table entry values so the page can be accessed. A third possibility is that the virtual page is an uninitialized page. Many programs allocate memory dynamically from a large uninitialized heap area in addition to uninitialized statically allocated data structures such as arrays. Instead of allocating physical memory for these uninitialized virtual
pages, the operating system simply sets the corresponding page table entries to zero-on-demand. The operating system often uses one of the software bits in a page table entry to indicate that the virtual page is zero-on-demand. Zero-on-demand means that the page should be allocated and zero filled the first time it is referenced. When the operating system detects a zero-on-demand page fault, it obtains a free frame, zeros out the contents of the frame, and sets the page table entry so the frame can be accessed. Some operating systems also allow for garbage-on-demand pages. Garbage-on-demand pages may be used if the program/application plans on initializing the pages itself and does not care what the contents of the page are. The fourth possible reason the hardware may think a virtual page is not in physical memory is if the page really is in memory, but not on the active list. A page may be in such a state if the operating system thinks the page will not be needed in the near future and moves it from the active list to the modified list (or to the free list). When the operating system moves the frame off the active list it invalidates the page’s hardware valid bit so the page can be written to secondary storage (or reclaimed) without fear of it being accessed. If a page is referenced that is currently on either of these lists, the hardware will signal a page fault. In this case, the page fault handler simply sets the hardware valid bit to true and moves the page back to the active list. We call this type of page fault an in-memory page fault.

The second most prominent reason a page fault may occur is because the user does not have the correct privilege to access the page. For example, this case may arise when a user attempts to write to a read-only page. The page fault handler usually reports a protection fault as an error. Protection faults are sometimes used to implement copy-on-write semantics which we will describe later.

3.3 Page Replacement

Up to this point we have said nothing about how the operating system decides which pages will be needed next (i.e., which pages to keep in physical memory). Deciding which page will be needed next is a difficult problem. In reality, the operating system
cannot know when a page will be needed. Instead it uses heuristics to make an 'educated guess' to decide which pages should stay in memory, and which pages can be written to secondary storage. If the heuristics fail, then a page fault occurs and the page must be fetched from secondary storage.

There are many page replacement algorithms an operating system can use to decide which pages should stay in memory. It is impossible to cover the wide range of page replacement algorithms in this paper, so we will simply present some of the concepts and policies underlying the various algorithms, and then take a closer look at one algorithm in particular.

Page replacement algorithms can be classified into one of two groups: global replacement, or local replacement. Global replacement algorithms choose the frame to replace from a 'global set' of frames (i.e. all the frames in the system). Local replacement algorithms, on the other hand, allocate a fixed number of frames for each ready process. When a process needs a frame, the replacement algorithm chooses a frame from the 'local set' of frames allocated to that process.

Global replacement has its advantages and disadvantages. The amount of memory needed by an application can change dramatically over the lifetime of the application. Because processes are allowed to steal frames from other processes, the number of frames an application may have at any given time can change to meet the application's needs. Applications requiring a large amount of memory can use frames not used by applications requiring a small amount of memory. On the other hand, if the number of processes in the system becomes too large, processes may spend all their time stealing pages from each other. This is known as thrashing, and occurs when none of the processes can get enough frames.

Local replacement also has advantages and disadvantages. Local replacement solves the thrashing problem that can arise with global replacement algorithms. Each process may only steal frames from itself, so an application cannot affect the performance of another application. However, because the amount of memory needed by an application can change dramatically over the lifetime of the process, it is difficult
to choose the correct number of frames to allocate to a process. If an application has not been allotted enough frames, it may thrash against itself. If an application has too many frames or is blocked in a queue, it wastes valuable memory space. Local replacement also limits the amount of multiprogramming allowed in the system. A process is only allowed into the system when there are enough frames available to meet its needs. This limits the number of process that can execute concurrently to the maximum number of processes that will fit in memory.

Some common global replacement algorithms are First In First Out (FIFO), Least Recently Used (LRU), Global-Clock (Second-Chance), Least Frequently Used (LFU), and Most Frequently Used (MFU)[PS85]. Most global replacement algorithms can also be used as local replacement algorithms. Local replacement algorithms include Working Set Replacement, Page Fault Frequency (PFF), and Working Size[Den80][Fin88]. Some of these algorithms are a little better than others because they use more information to make a better guess. However, they are also quite expensive to implement and may in fact require more hardware support than is available. Other algorithms only approximate the better algorithms; however, they can be implemented much easier, do not require special hardware, and are a close approximation. We will look briefly at one such algorithm, called the global clock algorithm.

The global clock algorithm is a global replacement algorithm that approximates the LRU algorithm. All the active frames are linked together, forming a circular list. A ‘clock hand’ routinely rotates around the circular active list (the dial) checking each frame it passes. A page is said to be “recently used” if an application references it before the clock hand can make a complete trip around the circular active list. A page is “not recently used” if the clock hand can make a complete trip around the circular list before the page is referenced again. The clock hand begins one frame past the frame that was last replaced. When a frame is needed, the clock hand advances until it finds a frames whose reference bit is not set and selects it for replacement. While the clock hand advances, it passes over frames whose reference bits are set. As it passes over these frames, it turns their reference bits off. If these pages do not get
referenced before the clock hand comes around again, the clock hand will find their reference bits off and will reclaim them.

3.4 Memory Sharing

In the introduction we mentioned that it is often attractive to share memory between application programs. This is especially true of object oriented applications which rely heavily on message passing and data sharing. The operating system can also reduce the total amount of physical memory used by allowing multiple instantiations of the same application (two processes running the same code) to share their code section.

The operating system provides shared memory by taking advantage of the hardware virtual memory support. Because the hardware translates every virtual memory address to a physical address, providing shared physical memory is actually quite simple. The operating system allows two or more processes to share the same physical page by setting the correct page table entry in each process to point to the shared physical page. A physical page can be mapped to different locations in different virtual address spaces. Moreover, a physical page may be shared by multiple processes. Each process sharing the page has a page table entry pointing to the shared physical page. Some operating systems even allow a physical page to appear twice in a single virtual address space. That is, two distinct pages in a virtual address space may share the same physical page.

The operating system data structures usually require some slight modifications to support shared memory. In particular, if a shared frame needs to be replaced, then all the page table entries that point to the physical frame need to be modified to reflect the fact that the frame is no longer in memory. We saw earlier that each frame table entry contained a pointer back to the page table entry associated with the frame, but now the frame table entry must maintain a list of all the page table entries associated with the frame. When the shared frame is about to be paged out, the operating system must invalidate each page table entry associated with that frame.
Invalidating the page table entries prohibits any of the processes sharing the frame from accidentally accessing the new data stored in the frame.

Copying arrays or other large data structures within a virtual address space or even from one virtual address space to another virtual address space becomes extremely efficient using shared memory and a technique known as copy-on-write. Rather than actually copying the frame, the operating system locates the page table entry corresponding to both virtual pages, the from page and the to page. It then sets these page table entries to point to the same physical page and sets the protection bits on both page table entries to read-only access. The copy operation is postponed until a write access occurs on one of the two virtual pages. When a write access occurs, the copy must be performed. Once the data has actually been copied, the page table entries are set to point to their own copy of the page, and the protection bits are reset for read/write access. Often data is copied from one process to another, but the second process only references (reads) the data. Copy-on-write makes this operation extremely efficient.

4 Secondary Storage

4.1 History of Secondary Storage

Computer systems have used numerous devices for secondary storage in the past. However, only a handful of these devices are suitable for supporting a virtual memory system. In order for a virtual memory system to perform 'reasonably', the operating system must be able to page frames in and out fast enough so that the user does not notice that it is happening. In other words, the virtual memory system performance must be comparable to the performance of non-virtual memory systems.

Originally, drums were the only device with fast enough access times to support paging activity. Eventually magnetic disk drives became the main secondary storage medium with seek and transfer rates fast enough to support paging. The operating system typically divides a disk into partitions. One or more of these partitions make up the swap partition. The paging system uses the swap partition as its secondary
storage, leaving all remaining partitions for the file system to use. Recently, remote file servers connected to a local area network (LAN) have become popular as a secondary storage medium. High speed LANs support data transfer rates fast enough to provide reasonable performance. Workstations view the file server as if it were a disk drive, the only difference being the fact that it must be accessed over the network. Each read, write, and seek operation is sent over the network to the remote file server rather than being sent to a disk controller. For all practical purposes, the remote file server acts like a disk drive, providing storage for both the file system and the paging activity.

We are investigating a new model for remote secondary storage. Our system separates the file system from the paging system by devoting a special page server machine to the task of paging. Unlike systems which page to a remote file system, our paging system does not have to compete with the remote file system for disk access. Furthermore, a remote file system behaves like a general usage raw disk with no knowledge about what is being stored on the disk. On the other hand, a dedicated page server machine understands the paging activity and the data being stored. As a result, the page server machine can use this knowledge to optimize paging performance.

4.2 The Page Server

The page server is a dedicated machine, handling paging requests from numerous diskless client machines (see figure 3). The performance of the entire system hinges a great deal on the performance of the page server. In order for the page server to support numerous client machines and still provide the performance we desire, the page server must be extremely efficient. In an effort to optimize the performance of the page server, we investigated methods that would minimize the page servers two most time consuming tasks: storing/fetching pages, and sending/receiving network packets.

To minimize the time spent storing/fetching pages, the page server machine em-
Figure 3: Dedicated page server machine serving diskless client workstations

<table>
<thead>
<tr>
<th>System Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
</tr>
<tr>
<td>Page Server</td>
</tr>
<tr>
<td>Disk</td>
</tr>
<tr>
<td>File Server</td>
</tr>
<tr>
<td>Other Servers</td>
</tr>
<tr>
<td>Gateway</td>
</tr>
<tr>
<td>LAN</td>
</tr>
<tr>
<td>Internet</td>
</tr>
<tr>
<td>Client</td>
</tr>
<tr>
<td>Client</td>
</tr>
<tr>
<td>Client</td>
</tr>
</tbody>
</table>

The page server employs a two-level backing store and a highly efficient page access algorithm. The first level of the two-level backing store is the physical memory on the page server machine, and the second level is the disk(s) connected to the page server machine. The page server caches as many pages as possible in memory for fast access and writes all remaining pages to disk. Because the page server knows about the paging activity, it can optimize system performance by using any one of the page replacement algorithms discussed earlier to decide which pages should stay in physical memory and which pages should be written to disk. The page server also uses a page access algorithm that can store/retrieve pages in almost constant time. Each store/fetch request is uniquely identified by a machine identifier, a process identifier, and a page number. 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 is no more than three. The combination of these two techniques allow the page server to handle most store/fetch requests in constant time.

To minimize the time spent sending/receiving pages, the page server uses a special purpose transport protocol between the client machines and the page server. The
details of the protocol are discussed in the following section.

4.3 Network Paging Protocol

The page server and its clients use a specialized paging protocol to minimize the time spent sending and receiving packets. Although conventional local area networks are reasonably reliable, packets still get dropped, delivered out of order, or arrive late. Every page store/fetch request must be delivered reliably and quickly to produce the performance we desire. To achieve this goal, we designed a paging protocol that is general enough to handle different page sizes from different architectures, yet it provides reliability and efficiency with low overhead and low delay.

There are two layers to the paging protocol. The upper layer deals with page store/fetch requests and end-to-end reliability, while the lower layer handles fragmentation. There are four message types the upper layer uses: page read requests, page write requests, process creation messages, and process termination messages. Since the upper layer assumes the underlying protocols and network are unreliable, it is the responsibility of the upper layer to provide reliability[SRC84]. When a page store or page fetch request is sent to the page server, the request is timed and reissued if an acknowledgement is not received in time. In the case of a page fetch request, the returned page acts as the acknowledgement.

Given a particular architecture, the page size may be much larger than the maximum network packet size. In order to transmit a page store/fetch request message, the message must be fragmented. The lower layer handles the fragmentation and also improves reliability. The upper layer guarantees reliability so that reliability is not needed at the lower level. However, a message may be broken up into numerous fragments. If any of the fragments is dropped or arrives late, the entire message (all the fragments) must be retransmitted. To improve reliability, the lower layer uses a low overhead negative acknowledgement protocol to detect errors early and attempt to correct them. Each fragment is assigned a sequence number and sent in sequence. As packets arrive on the receiving end, they are reassembled but are not acknowledg-
edged. The receiving end remembers the sequence number of the last fragment for each store/fetch request currently arriving. As soon as a fragment is received out of sequence, a negative acknowledgement containing the sequence number of the missing fragment is sent to the lower layer of the sender. The sender then resends the missing fragment again. In the expected case, where no packets are lost, there is no overhead involved at all. In the rare case when a packet is lost, only an error message for the lost packet is sent. Optimizing for the expected case makes the protocol very efficient. In addition, the lower layer improves reliability, avoiding the 'high cost guaranteed reliability' provided by the upper layer.

4.4 A Prototype Implementation

We have developed a virtual memory system that pages to a remote page server machine over a local area network. The client operating system is based on the XINU operating system[Com84] [Com87], and the prototype page server is a single UNIX\(^6\) process rather than a dedicated machine. The client machines are SUN3/50s and Digital Equipment Corporation MicroVax II's running a virtual memory version of XINU[Gri89]. The page size on the MicroVax is 512 bytes while the page size on the SUN is 8K bytes, yet both architectures page to the same page server simultaneously. The page server runs on either a SUN3/50 with 4 Mbytes of memory, or a VAX 11/780 with 8 Mbytes of memory and is connected via a 10 Mbit/sec Ethernet[Dig80] to the client machines.

Virtual Memory Xinu uses the global clock algorithm described earlier for page replacement. It also supports shared memory by allowing pages to be physically shared between processes. Contention and synchronization for use of the shared segments is handled with the use of semaphores.

Our paging performance results are very encouraging. The average time to service a page store/fetch request on a UVAX II (512 byte pages) is 10-20 milliseconds, which is quite reasonable when compared to the average seek time of 10-16 milliseconds for

---

\(^6\)UNIX is a registered trademark of AT&T Bell Laboratories.
a conventional disk drive, the average time to service a page store/fetch request from a SUN with a much larger page size (8K bytes) is 40-50 ms. Considering that the prototype page server machines have relatively small physical memories by conventional standards, these times, as well as the system performance on the client, seem quite reasonable.

Virtual Memory XINU also allows us to set a software page size, independent of the physical page size of the machine. To demonstrate the effect of using various page sizes, we ran three tests on the UVAX II and varied the software page size. The first test simply ran through the heap area over and over sequentially reading/writing every 512th byte. The second test accessed random locations in memory. The third test simulated an application program accessing code, data, heap. The results appear in Table 1.

<table>
<thead>
<tr>
<th>Test Program</th>
<th>0.5K page</th>
<th>1K page</th>
<th>2K page</th>
<th>4K page</th>
<th>8K page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>109</td>
<td>75</td>
<td>50</td>
<td>34</td>
<td>29</td>
</tr>
<tr>
<td>Random</td>
<td>38</td>
<td>75</td>
<td>99</td>
<td>134</td>
<td>310</td>
</tr>
<tr>
<td>Sim. Program</td>
<td>96</td>
<td>86</td>
<td>84</td>
<td>81</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 1: Total Run Time (seconds) For Varying Page Sizes

5 Summary

Conventional workstations now perform many of the task previously only possible on mainframes and minicomputers. Workstations are also able to support new applications like windowing systems, WYSIWYG editors, and object oriented applications. Workstation operating systems provide a virtual memory abstraction used to support the large amounts of memory needed by these new applications.

We showed how the operating system creates the virtual memory illusion by paging portions of memory to and from secondary storage. We outlined how the operating system uses data structures, algorithms, and hardware support to provide each application with its own virtual address space. Finally, we described how diskless
workstations using a special-purpose high speed protocol can use remote machines for secondary storage and still attain acceptable performance.

References


