CMU-CS-85-164

The Integration of Virtual Memory Management and Interprocess Communication in Accent

Robert Fitzgerald and Richard F. Rashid
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

16 September 1985

This is a preprint of a paper scheduled for publication in the May 1986 edition of ACM Transactions on Computer Systems. Some changes may be made before publication.

Abstract

The integration of virtual memory management and interprocess communication in the Accent network operating system kernel is examined. The design and implementation of the Accent memory management system is discussed and its performance, both on a series of message-oriented benchmarks and in normal operation, is analyzed in detail.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

List of Figures

List of Tables

1. Introduction

All communication-oriented operating systems share the problem of getting data from one process to another. System designers have traditionally chosen one of two alternatives:

1. processes pass data by reference

2. processes pass data by value through the exchange of messages

By-value message systems typically require that message data be physically copied. If the semantics of the message system allow the sending process to be suspended until the message has been received, only one copy operation is necessary. Alternatively, explicit primitives can be provided that allow data to be copied once from the sender to the receiver after the message has been received [4, 8]. Asynchronous message semantics often require, however, that all message data be copied twice: once into a kernel buffer and again into the address space of the receiving process [20, 30].

Not surprisingly, data copying costs can dominate the performance of by-value message systems [2]. The CPU cost of copying data can be high and the memory storage costs, the result of having to make two or three separate copies of the data transmitted, can also be important, as can costs of paging the data into memory where it can be copied. Such systems often limit the maximum size of a message, forcing large data transfers to be performed in several message operations [7, 20].

In systems that allow by-reference sharing of memory, processes may either share access to specific memory areas or entire address spaces. Messages are used only for synchronization and to transfer small amounts of data, such as pointers to shared memory. Communication between processes within a THOTH team [7] is an example of this approach.

By-reference sharing of data is much cheaper than copying for large data transfers on a single machine, but can seriously compromise system reliability and security. Several capability-based systems [15, 16, 34] have partially addressed this problem by passing memory access capabilities in messages. However, these systems do not address the problems of unintended or unsynchronized access to shared data. In addition, it is difficult and expensive to extend a by-reference memory access scheme transparently into a network environment [31].

In 1981, we began to implement Accent [27], a communication-oriented operating system kernel designed to support the needs of a large network of personal computers. One of the Accent design goals was that its communication abstractions be transparently extensible into the network

environment. We therefore chose to pass all data between processes by-value in messages. At the same time, experience with previous operating systems, notably Rochester's RIG system [20], led us to seek an alternative to data copying for large messages.

Our approach was to combine virtual memory management and interprocess communication in such a way that large data transfers could use memory mapping techniques rather than data copying. By-value semantics are preserved by transferring message data with copy-on-write memory mapping, so that both the sending and receiving process have their own logical (if not disjoint physical) copy of the data.

Through this integration of paged virtual memory management and interprocess communication, we hoped to achieve several goals:

- simple communication semantics, easy to understand and easy to extend into a network or a large multiprocessor
- simple process access to data such as files through memory mapping (as in traditional P-MAP style file mapping [6, 23, 28])
- the ability to transfer data objects in their natural size, up to and including the size of a process address space, unhindered by artificial message size limits
- better utilization of physical memory and backing storage through greater sharing between processes
- flexible transfer of data over a network through copy-on-reference network communication.

It has now been more than four years since we began to implement Accent as part of the CMU SPICE project [9]. Accent[1] is now (August 1985) running on a network of approximately 200 personal computers at CMU and is marketed commercially by PERQ Systems Corp. and Advent Ltd. with an installed base of over 1000 systems.

In addition to network operating system functions such as distributed process and file management, window management and mail systems, several applications have been built using Accent's primitives. These include research systems for distributed signal processing [14], distributed speech understanding and distributed transaction processing [32]. Four separate programming environments have been built -- CommonLisp, Pascal, C and Ada -- including language support for an object-oriented remote procedure call facility [18]. A commercial version of UNIX System V has even

---

[1] Accent is a trademark of Carnegie-Mellon University.

been built as an application on top of the Accent kernel [26].

As is often the case in a research environment, the original Accent design was largely based on intuitions about how best to integrate virtual memory and interprocess communication and the ways in which system facilities would be used. We now have sufficient experience with the system that we can judge how accurately our intuitions match reality.

In this paper we examine the way in which virtual memory management and interprocess communication are integrated in the Accent design and implementation. We will also look at how well the Accent performs its functions on a single machine by answering the following questions:

1. What are the costs associated with copy-on-write data transfer?
2. How much data is transferred in normal operation?
3. How are physical memory and backing storage utilized?
4. What is the relative frequency of use of system facilities?
5. What features of the system are cost bottlenecks in the current implementation?
6. How does the use of virtual memory with message passing affect the design and implementation of applications?

2. The design and implementation of Accent

The Accent kernel supports four basic abstractions:

1. *A* **message** *is a typed collection of data objects consisting of a fixed size header and a variable length body.*

   A message may be any size and may contain typed pointers to data outside the contiguous portion of the message.

2. *A* **port** *is a kernel protected queue for messages.*

   At any given time, the maximum length of a port is fixed, although that fixed length can be changed. Processes refer to ports through port capabilities. There are three kinds of port capabilities: send access, receive access and ownership. Processes obtain capabilities to ports only by receiving such capabilities in messages.

3. *A* **process** *is a thread of control operating in a 2**32 byte paged address space.*

   Processes may send and receive messages according to their access rights. When a process is created, the kernel also creates a port, called the ***kernel port*** of the process, to represent it. The state of a process and its virtual memory can be manipulated by sending messages to its kernel port. By default, only a process and its parent have access to its kernel port.

4. *A* **memory object** *is a kernel provided repository for data.*

   Memory objects can be created, destroyed, read or written. Backing storage for a memory object is determined by its type: permanent disk, temporary disk, physical memory or port. Permanent disk memory objects are used to manage files in the SPICE file system [17]. Temporary disk objects are used to back newly created virtual storage on disk and to shadow copy-on-write data (see section 2.2.2). Physical memory objects are used to manage devices that operate on physical memory. Port memory objects provide copy-on-reference network access to data [35] and any other on-demand creation or control of information.

The Accent kernel can itself be viewed as a process with its own 2**32 byte paged virtual address space and port access rights. The primary purpose of the Accent kernel is to provide an execution environment for user processes and an interprocess communication facility. The SPICE network operating system is implemented as a collection of such processes running above the Accent kernel using the Accent IPC facility to communicate. Port capabilities are used to represent process-provided services, resources and data structures. As such, port capabilities serve a role in Accent similar to object capabilities in systems such as Hydra [34] or STAROS [16]. Interprocess interfaces in Accent are defined using an object-oriented interface definition language called MatchMaker [18]. These interfaces are compiled into remote procedure calls (RPC) stubs that use the Accent message passing primitives for communication and control (similar to those built for Pilot [5]).

Accent is a single machine operating system kernel in the sense that all of its operations are defined to operate on a single processor. The kernel's IPC facility, for example, supports only communication between processes on the same machine. The key to Accent's use as a network operating system kernel is the fact that its abstractions, its IPC facility and virtual memory support, are designed to be transparently extensible by user-state processes. This permits traditional operating system functions to be provided by server processes -- which are typically easier to prototype and develop than an operating system kernel.

One example of this flexibility is the way Accent provides for network communication. Rather than provide kernel support for networking, network server processes have been implemented which transparently extend Accent's IPC facilities between machines. Another example of this use of Accent primitives to build more traditional operating system functions can be found in the file system. The kernel's memory object facilities are its only file support. A file server process builds user file abstractions, such as directories, on top of the Accent memory object. The file system server also uses network-transparent IPC to cooperate with other file servers in providing network-transparent remote file access. Process creation and destruction, a third example, is considered in detail in section 3.7.

2.1. The semantics of message passing

The underlying model of communication in Accent is asynchronous: a process sends a message to a port and sometime later a process with receive access to that port receives it. There is no explicit connection between sending and receiving events, so messages must be stored somewhere while they are in transit. Messages are potentially large, they may contain pointers, and their data must be passed by value. The receiving process may not know the size or structure of the incoming message in advance. Address map manipulation may be used to transfer large amounts of data. The design of the Accent IPC and virtual memory subsystems must thus address several issues:

- What do the memory mapping primitives do?
- How is data stored between the time that it is sent and the time that it is received? When is data shared? When is it copied?
- How does a receiving process specify where in its address space to put an incoming message?

2.1.1. The semantics of copy-on-write mapping

A single primitive called ***MoveWords*** is used within the Accent kernel to perform all memory mapping operations, whether to rearrange memory within an address space or to transfer it between address spaces during message passing. A call to MoveWords has the form:

```
MoveWords(SourceProcess, SourceAddress, DestinationProcess, DestinationAddress,
          NumberOfWords, DeleteSource, CreateDestination)
```

SourceProcess and SourceAddress specify the internal kernel process number and 32-bit starting address of an area to be transferred. DestinationProcess and DestinationAddress specify the process number and 32-bit starting address of the area to which the data is to be transferred.

DestinationAddress is an *inout* parameter whose use depends on CreateDestination. If CreateDestination is true, the DestinationAddress provided is ignored and a new area of size NumberOfWords is created in the address space of process DestinationProcess. If necessary, up to a page of padding is added to both ends of the new area so that the offset of the area within a page is preserved and the data can be remapped rather than copied. The address of the new data area is returned in DestinationAddress.

If DeleteSource is false, the source data is shared copy-on-write between source and destination. Otherwise, the data is moved from source to destination by removing the source access to the data as destination access to it is added. Since no new sharing is introduced if the source is deleted, the data is not protected copy-on-write.

MoveWords is used to transfer pointer data into and out of the kernel. When copying data into the kernel during the message Send operation, SourceProcess is the sending process and DestinationProcess is the kernel. When copying data out of the kernel during the message Receive operation, SourceProcess is the kernel and DestinationProcess is the receiving process.

A process can use MoveWords to manipulate its own address space. A kernel trap provides access to MoveWords with the trapping process as both SourceProcess and DestinationProcess.

A process can also use MoveWords to transfer data between its address space and the address space of a process identified by its kernel port. The ReadProcessMemory and WriteProcessMemory kernel calls are used to initialize process address spaces (see section 3.7.3) and by debuggers to peek and poke in target process address spaces.

2.1.2. Storing messages

A message is divided into three parts:

1. a ***message header*** of fixed size and format
2. a variable-size block of ***in-line data*** contiguous with the header
3. any number of blocks of ***pointer data*** consisting of virtual memory pointed to by the in-line part of the message

Messages are stored in the virtual address space of the kernel while in transit. The kernel maintains a linked list of free buffers used to hold messages in transit. The buffer size, fixed at kernel compile time, is large enough to hold a message header, a moderate amount of in-line data, and a few bytes of overhead[2]. The header and small in-line data blocks are copied into and out of the kernel. Larger amounts of in-line data and any pointer data are mapped into and out of the kernel and are stored in kernel virtual memory that is dynamically validated by the MoveWords CreateDestination option.

The sending process can specify that individual pointer data blocks be deallocated by the MoveWords DeleteSource option as they are mapped into the kernel. The header and in-line data portions of the message cannot be deallocated as a direct result of the send operation but can be subsequently deallocated by the sending process. The kernel uses DeleteSource to deallocate large in-line data and pointer data blocks as they are transferred into the receiving process during the receive operation.

Figure 2-1 illustrates the transfer of a message containing a pointer data block. Process A sends the message to port P1, from which process B receives it. Valid memory is shown with a solid outline in the process maps, invalid memory with a dotted outline. The pointer data is shown as a cross-hatched area in the map of processes with access to it. The pointer data is duplicated into the kernel address space during the send operation and moved from the kernel address space to process B's address space during the receive operation. Note that the data has then been deleted from the kernel address space, but is still shared between processes A and B.

No new physical memory or backing store is normally used to hold mapped in-line data or pointer data, as the data is shared copy-on-write between sender and kernel. Should the sending process modify pages of copy-on-write data before they have been received, the kernel makes copies of those pages (by handling CopyOnWriteCopy faults, section 2.2.5) to preserve their original value. Backing

---

[2]The current buffer size is 1024 bytes, which accommodates the 22 byte header and 960 bytes of in-line data.

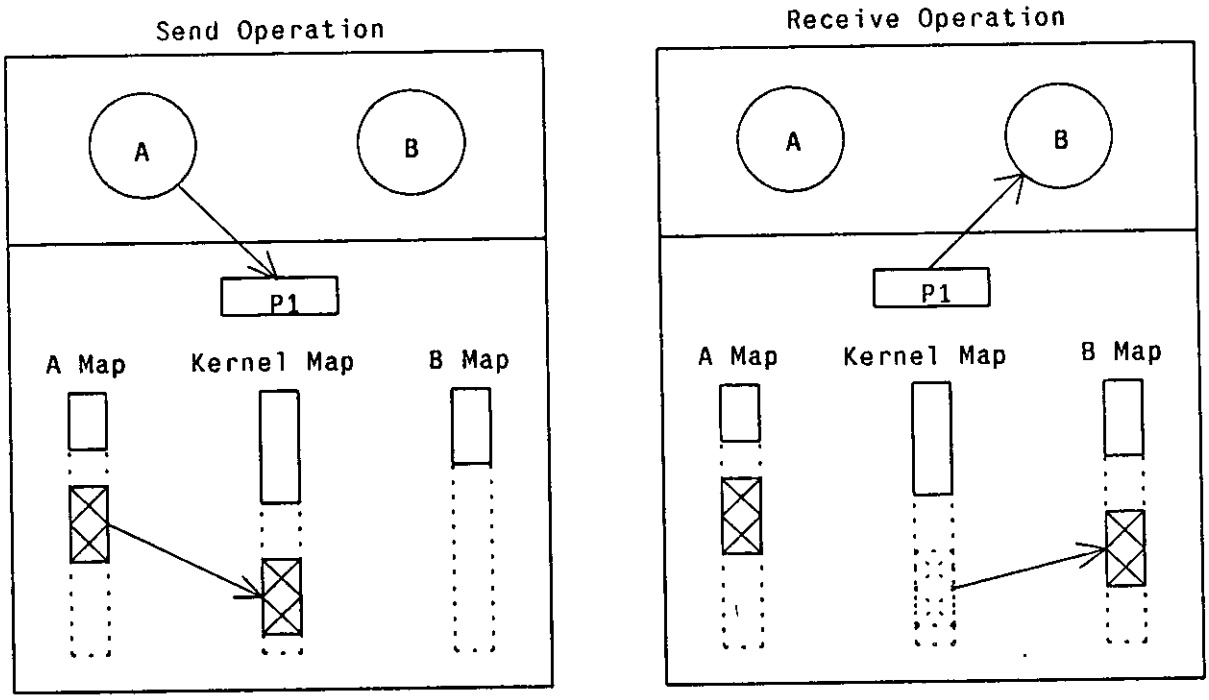

Figure 2-1: Memory mapping operations during message transfer

store for these copied pages is not allocated until they must be swapped out.

Storing messages in the kernel while they are in transit has both advantages and disadvantages. The major advantage is simplicity: a single mechanism (MoveWords) can be used for all copy-on-write mapping operations. The major disadvantage is the introduction of an arbitrary implementation restriction: a limit on the total amount of virtual memory that can be in transit at the same time. This has not been a problem to date, as the kernel has the greater part of a gigabyte of address space in which to store data in transit, and few messages are typically in transit at once. Another disadvantage of this scheme is that storing data in the kernel implies the cost of transferring data both to and from the kernel address space. Other schemes, such as Walden's IPC [33], permit data to be transferred only once, directly from source to destination.

2.1.3. Receiving a message

The receiving process specifies the location in its address space for the header and contiguous data portion of the message. If insufficient room is provided, the kernel returns an error code to the receiving process. Pointer data in the message is transferred by MoveWords with the CreateDestination and DeleteSource options, i.e. to arbitrary unused virtual memory in the receiver, deleting the kernel access to the memory during the transfer.

Receiving processes have no control over where pointer data in messages is placed in their virtual memory. Should a receiving process want that data to be placed at a specific location, it must use the MoveWords trap to move it there after the message has been received. Few processes do so in practice, and the advantages of allocating new virtual memory for the data outweigh the disadvantages.

2.1.4. Message semantics: an example

The way in which Accent processes read and write files illustrates both the semantics of message passing and the effect of copy-on-write data transfer on application programs.

File access in Accent is mediated through a file server process. This approach is similar to that of other message based systems, notably the VKernel [8] and RIG [20]. The file server determines the access rights of processes making file requests and translates those requests, if valid, into operations on Accent permanent memory objects. Unlike other message systems, the Accent file server does not physically handle the data contents of files or maintain state about the handling of file data. Files in Accent are treated as values and are passed by copy-on-write mapping.

For example, if a file system client process wants to read the contents of the file "*README*", that client sends a message to the file server that means: "Give me the contents of file *README*". The file server maintains directory data structures describing the mapping between file names and corresponding permanent memory objects. If the file *README* exists and if the client has read access to it, then the file server in turn sends a message to a port serviced by the Accent kernel (representing access to permanent memory objects) requesting the contents of the permanent disk memory object corresponding to *README*. In reply to that message from the file server, the Accent kernel sends back a message that contains the contents of *README* copy-on-write. The file server receives that message containing *README* and can then forward the contents of that message to the client. When the client process receives the reply from the file server, the file *README* has been mapped into the client's address space with copy-on-write protection and can be referenced by the client as ordinary virtual memory.

In this example, only four messages were sent, two containing pointer data. The data in the file *README* on disk was not accessed. Yet, *README* has been mapped into the client's address space and can be referenced accordingly. By-value message passing semantics have been preserved at a cost similar to that of accessing file data with a P-MAP file mapping primitive.

Copy-on-write protection prevents the client from accidentally modifying the file README on the

disk, thereby also avoiding problems of shared file and memory semantics in distributed systems. Client processes intentionally write disk files by reversing the file reading process. The client sends a message to the file server asking it to "write this data to the file WRITEME". The file server asks the Accent kernel to write the data to the permanent disk memory object corresponding to WRITEME. The data is transferred by copy-on-write memory mapping in both cases and is ultimately copied to the permanent disk object by the Accent kernel. Should any process need continued access to the old value of WRITEME, the kernel creates a shadow object (see section 2.2.2) during the store into the permanent disk object. This shadow object containing the old value then backs any virtual memory previously backed by WRITEME.

2.2. The design and implementation of memory management in Accent

The requirements of memory-mapped message passing made it clear from the outset that the Accent virtual memory system had to support reasonably frequent copy-on-write mapping of data. In addition, the desire to support a wide variety of virtual memory intensive applications (e.g., AI software built in CommonLisp) dictated compact, easy to manipulate process maps that could support sparse use of an address space.

The Accent virtual memory system, like that of the Apollo Aegis [21] and IBM's System 38 [12], is built on the notion of single-level store. Primary storage is used as a cache of secondary store, which is in turn organized around the concept of the Accent memory object. Logically, a page in the address space of a process is mapped, not directly to disk, but instead to a page of a memory object that may be backed on disk. Memory objects provide a level of indirection that permits a variety of media to back virtual memory.

2.2.1. The Accent memory object

A particular Accent memory object is identified within the kernel by a 32-bit identifier that determines its type: ***permanent disk***, ***temporary disk***, ***physical memory*** or ***port***. Memory objects in active use are represented in a core-locked Active Segment[3] Table (AST) that specifies how to find their contents. This is the only header information kept for transient memory objects. The header information for permanent disk objects is also stored on disk to allow such objects to persist across system reboots.

For permanent disk memory objects, the memory object identifier is the disk address of the header

---

[3]The entities called memory objects in this paper are known to the Accent kernel code as segments. The name "segment" was dropped from the paper because it evoked too many images of two-dimensional virtual memory, with its alignment and size restrictions that Accent doesn't suffer from.

that contains information about the object. To protect against corruption of the disk data structures due to single drive or controller failures, each disk block contains, in addition to its data, an eight word header that contains the block's object identifier, logical offset within that object, and links to the previous and next logical blocks of the object. The object header includes the head of a random index of the object's disk blocks and the head and tail of a linked list of those blocks. Three sources of information can thus be used to verify that a block belongs to a memory object: the random index in the object header, the linked list of blocks in the object, and the object identifier in the header of the block itself.

Temporary disk objects do not require the kind of reliability (and implied costs) built into the permanent disk store. Disk space for temporary disk objects is allocated from a special paging partition from which pages can be allocated and deallocated more cheaply. The memory object identifier for a ***solid*** object is the disk address of a contiguous range of logical disk blocks. The memory object identifier for an ***indexed*** object is the disk address of a random index for the disk blocks of data. Solid objects are used to back small regions of newly created memory. Indexed objects are used to back larger regions and for copy-on-write shadows.

Physical memory objects are backed by a contiguous region of physical memory whose address is the memory object identifier. Physical memory objects should not be confused with the pool of physical pages used to cache pages of virtual memory.

Port memory objects resolve to a data structure containing information about the size and status of the object and a port capability. Faults and attempts to swap pages in or out are transformed into messages.

2.2.2. Shadow memory objects

Copy-on-write sharing of data in memory objects requires that objects be able to share some, but not all, of their data with other objects. This need is met by ***shadow*** objects, which have some pages of their own and share others with the object that they shadow, which may in turn be a shadow object. A shadow object is always an indexed temporary disk object that contains a pointer to the object it shadows and only the pages that differ from that object. Pages in a shadow object are found by looking first in the shadow, then in the object it shadows, etc. until a page with the proper offset is found. Figure 2-2 illustrates a two level shadow.

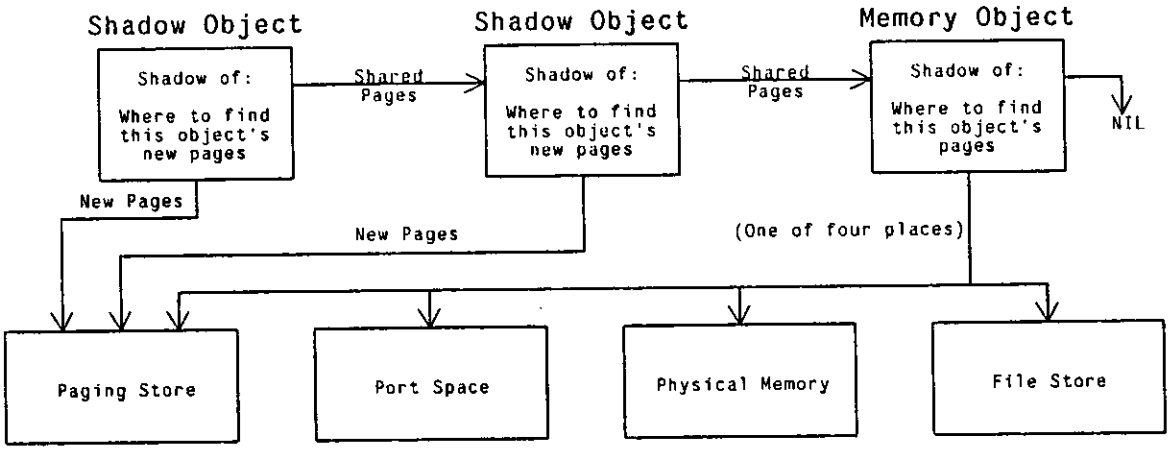

Figure 2-2: An example of memory object shadowing

2.2.3. The Accent process map

Accent maintains a process map for each user process and one for the operating system kernel. The kernel's address space is paged and all user process maps are kept in paged kernel memory. Only the kernel process map, a kernel stack, kernel static variables and those kernel code modules required for handling the simplest form of page fault are locked in physical memory.[4]

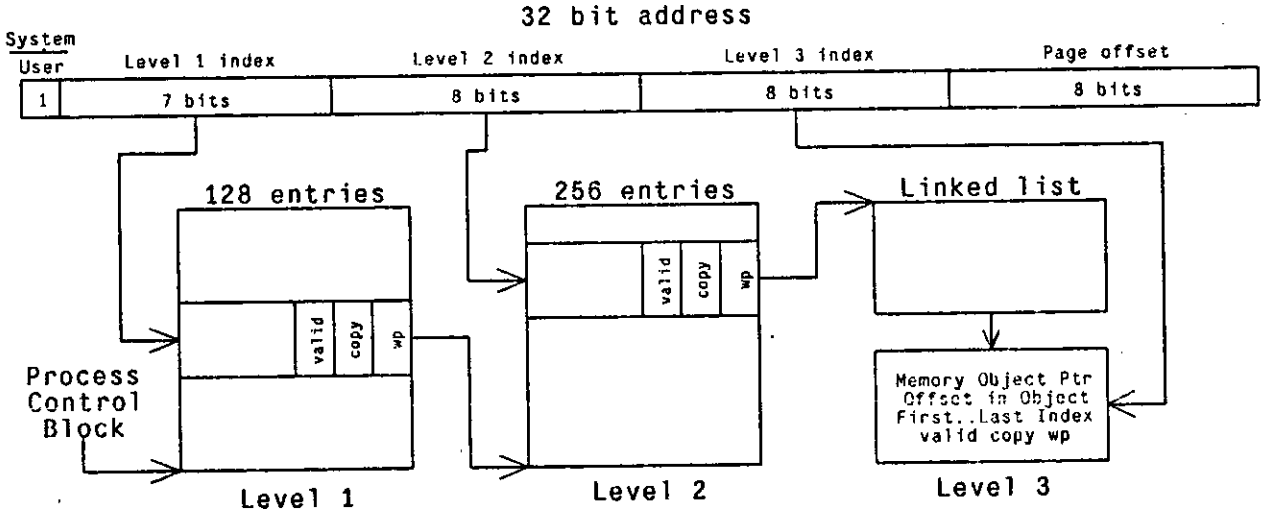

Figure 2-3: Mapping a virtual address in Accent

The Accent process map (figure 2-3) is 3 levels deep. The top two levels are indirect tables while the third level is a linked list of entries that map a range of contiguous process virtual pages into contiguous regions of Accent memory objects. The process map is organized so that large portions

[4]Although most of the kernel code is pagable, it is expedient to lock it down during kernel development. Most measurements in this paper were made with all kernel code locked.

can be validated, invalidated or mapped without having to modify the linked lists of map entries.

The level-1 table is pointed to by the process control block and consists of 128 entries, each mapping 32 megabytes of data. Individual entries consist of status bits and short pointers to level-2 table blocks. Level-1 status bits can indicate that the 32 megabyte area represented by the mapping entry is valid, copy-on-write or write-protected.

Individual level-2 table blocks contain 256 entries, each mapping 128 kilobytes of data. These entries consist of status bits and short pointers to a linked list of level-3 ***chunk descriptors***. Level-2 status bits can indicate that the 128 kilobyte area represented by the mapping entry is valid, copy-on-write or write-protected.

Chunk descriptors are 16-byte data structures describing the backing of a contiguous range of process virtual pages by an Accent memory object. Each contains the first and last level-3 indices that it describes, a pointer to the Accent memory object that backs the page range, and a page offset in that object to the first page described by the chunk descriptor. A chunk descriptor also contains status bits indicating whether the page range that it describes is valid, copy-on-write or write-protected. The linked list of chunk descriptors is not ordered but in actual use is seldom very long, often consisting of zero, one or two descriptors.

Accent process maps are kept in compact regions of kernel virtual memory. This promotes locality of reference and makes creation and destruction of entire address spaces simpler and cheaper. It also allows an entire map to be duplicated in a fork operation using the copy-on-write mapping facilities of Accent.

A typical Accent process (the Process Manager of the CMU standard distribution system) with over 500K bytes of code and data spread over 2 megabytes of address space requires a minimum map size of only 1024 bytes. A CommonLisp core image of 8 megabytes of code and data spread over four gigabytes of address space needs a map of only 12K bytes. A comparable VAX 11/780 map would require 16 megabytes of kernel virtual memory. Even assuming a compact address space, 8 megabytes of data would result in a VAX map size of 64K bytes, over five times the size of the Accent map.

2.2.4. The VP table

The Accent process map is neither intended for nor suited to paged address translation. Its compactness makes lookups expensive, and it makes no attempt to represent individual pages. A second structure, called the ***Virtual to Physical address translation table*** or ***VP table***, is used for address translation and to keep track of the mapping between process virtual addresses and physical memory. This table is hashed, with process number and virtual address as its key (a scheme similar to that used in the PRIME Corporation PRIME 750 [25]). The size of the VP table depends on the size of physical memory. It currently needs at least twice as many entries as there are physical pages, and performs better with four times as many.

The contents of the VP table supersede the information in the Accent process map. This allows changes caused by zero-fill page allocations and copy-on-write updates, for example, to be recorded first in the VP table (locked in physical memory) and only incorporated into a process map when the relevant page must be written out to secondary storage (an already expensive operation).

Disk space is not allocated to back up a process address space until the page must be written out to disk. When an unbacked physical page must be paged out, Accent allocates temporary object backing for it, either by adding a new page to an existing memory object or by creating a new memory object.

If a page to be swapped out is backed by a memory object with copy-on-write protection, then the memory object cannot be modified. A temporary disk object is allocated to be a shadow of the underlying memory object and the process map is updated to refer to the shadow object. The page is then written out to the shadow object.

2.2.5. Handling a page fault

When a process takes a fault, it is blocked onto a fault-pending queue serviced by the ***Pager/Scheduler***, a special core-locked, supervisor-state-only process that handles all faults. Virtual memory fetch operations cause ReadFaults. Virtual memory store operations cause WriteFaults. Both Read- and WriteFaults are qualified by the process and virtual address at which they occur.

Both Read- and WriteFaults are AddressFaults if there is no VPTable entry for the faulting process at the fault virtual address. AddressFaults are further divided into AddressErrors, ZeroFillFaults, MemoryFillFaults and DiskFillFaults.

An ***AddressError*** occurs if the fault virtual address is not valid in the address space of the faulting

process. The faulting process is suspended and a message is sent to a guardian process, normally the ProcessManager, to notify it of the error. The guardian process typically responds to the notification by invoking a debugger process on the faulting process.

A ***ZeroFillFault*** occurs if a valid fault virtual address is neither cached in physical memory nor backed on disk[5], typically when a previously unused virtual page is first referenced. The ZeroFill fault completes the lazy allocation of physical caching for valid memory. A new physical page is allocated and zeroed, and a mapping from the fault virtual page to the new physical page is entered in the VPTable. Disk backing for the virtual page is not allocated at this time and the map of the faulting process is not modified.

A ***MemoryFillFault*** occurs if a valid fault virtual address is backed by a disk page that is physically resident. A mapping from the fault virtual page to that physical page is entered in the VPTable.

A ***DiskFillFault*** occurs if a valid fault virtual address is backed by a disk page that is not physically resident. A new physical page is allocated, the disk page read into it, and a mapping from the fault virtual page to the new physical page entered in the VPTable.

ReadFaults only cause AddressFaults. WriteFaults can require additional handling beyond a possible AddressFault. The AddressFault handler has already made the fault address physically resident and readable if it hadn't been already. The extra WriteFault handling is divided into WriteErrors and CopyOnWriteFaults.

A ***WriteError*** occurs on an attempt to store into a ReadOnly (aka WriteProtected) page. It is handled like an AddressError, except that the error notification message indicates a protection violation.

A ***CopyOnWriteFault*** occurs on an attempt to store into a copy-on-write page. ***CopyOnWrite*** protection indicates a page that once contained a shared value. If the physical page is no longer shared, i.e. it has only a single VPEntry mapping to it, the protection of that virtual page is changed from CopyOnWrite to ReadWrite. This is a ***CopyOnWriteReclaim***. To reduce the fault handling overhead of removing CopyOnWrite protection from pages that are no longer shared, a single fault also removes CopyOnWrite protection from succeeding unshared virtual pages.

A CopyOnWriteFault on a virtual address that maps to a shared physical page is a

[5]Strictly speaking, virtual memory is backed by an Accent memory object. This description uses 'disk' to be concrete.

CopyOnWriteCopy. The fault completes lazy evaluation of the copy operation postponed by a copy-on-write mapping operation. A new physical page is allocated, the old physical page copied into it, and the mapping of the fault virtual page in the VPTable changed from the old physical page to the new one. Other virtual addresses that previously mapped to the old shared physical page continue to do so.

CopyOnWrite faults, like ZeroFill faults, do not cause allocation of disk backing or modification of the faulting process' map.

2.2.6. Page replacement

The extensive and often unpredictable interdependencies between client and server processes in Accent (and the simple implementation) led us to use a global rather than process-local page replacement strategy, as did Unix 4.1bsd [1]. Accent uses a global LRU approximation related to the classic global clock algorithm [10, 11]. This basic algorithm is complicated by Accent's single-level store nature and its lazy evaluation of process map changes and backing storage allocation. These complications cause special handling by page replacement for two categories of physically resident pages: memory object pages with no process mapping and process pages with no memory object backing.

Memory object pages that are not mapped into the address space of a process occur primarily because idle physical memory is used as a disk cache, permitting recently used pages to be found in memory without reading them in from disk again. Such cached disk pages are assumed to be likely to be referenced than process-mapped pages and are aged at twice the rate of other pages. This accelerated aging also compensates in part for an anomaly that sets the used bit of a page when a VP entry mapping to it is removed.

Pages without memory object backing occur because Accent lazy-allocates backing store. Such unbacked pages present a problem because they cannot be written out to disk until new backing store and potentially even new memory objects have been allocated for them. This allocation requires locks that the kernel may be unable to acquire when it wants to swap out an unbacked page. As a result, the ability of the system to respond to a page replacement request can be impaired if unbacked pages occupy too much physical memory. Pages with disk backing can be paged out without disk allocation. Accent therefore guarantees that at least 1/4 of the physical pages available for paging are either free or backed on disk. This ensures that there will be enough disk-backed memory available to handle critical page allocation requests when disk allocation is impossible. This limit is rarely approached in practice, as most of memory is typically filled with cached disk pages.

3. The performance of Accent

In order to present a clear overall picture of the performance of Accent under different circumstances, we report performance information collected from three different sources:

1. A series of artificial tests were devised to provide detailed breakdowns of the time spent executing small scale primitive operations. These timings relate costs back to aspects of the design discussed in section 2.

2. A perspective on actual large scale use of Accent was obtained by examining the task of generating the entire system from sources stored on a file server in our local network. The distribution of costs for this task demonstrates that Accent can deliver acceptable overall performance. The way facilities and resources were used during this task reflects Accent's approach to resource management.

3. A special task, midway in scale between an artificial test and a large scale application, was constructed which consisted of repeated execution of a simple program. This task is large enough to be realistic yet small enough to be examined in great detail. It makes heavy use of Accent's IPC and virtual memory management primitives.

3.1. The experimental apparatus

All reported measurements were made on versions of Accent running on one of two PERQ Systems Corporation PERQs [24]. The PERQ is a microcoded 16-bit minicomputer with a microengine based on the AMD2910. It executes one 48-bit microinstruction from a writable control store every 170 nanoseconds. Memory is 16-bit word addressable, with a typical memory reference costing about 1 microsecond. This does not include the cost of virtual-to-physical address translation, for which there is no hardware support. Both PERQs are configured with 2Mb physical memory, 16K instruction WCS, hard disk, ethernet interface, keyboard, bitmapped display and pointing device.

The Accent kernel on the PERQ is written in a dialect of Pascal called PERQ Pascal [3]. This Pascal is implemented by a byte-encoded instruction set similar to USCD P-Code, interpreted in microcode at an average of 0.5 million byte codes per second. An average line of kernel source executes in approximately 20 microseconds. The Lampson 'tick' time [19], defined to be 1/4 of the time required to execute 'a := b + c', varies with the size and storage class of the variables used, from 1.25 microseconds (16-bit stack locals) to 2.3 microseconds (32-bit stack locals) to 4.1 microseconds (16-bit static variables).

Table 3-1 compares the relative performance of PERQ and VAX-11/780 CPUs. Timings were performed in Pascal on the two PERQs described and in C on a VAX running UNIX 4.1bsd. We chose this comparison because Vax/Unix4.1bsd is a widely known uniprocessor system of about the same age as Accent.

Perq	Vax	Ratio	Operation
2300ns	720ns	.31	Tick (32-bit stack local)
12us	4us	.25	Simple loop (16-bit integer)
20us	3us	.17	Simple loop (32-bit integer)
35us	20us	.57	Null procedure call/return
75us	25us	.33	Procedure call with 2 arguments
80us	400us	5.00	Context switch
132us	264us	2.00	Null kernel trap
30s	9s	.30	Baskett Puzzle Program (16-bit)
50s	10s	.20	Baskett Puzzle Program (32-bit)

Table 3-1: Comparison of Perq and Vax-11/780 operation times
For executing kernel code, a Perq CPU is about 1/5 of a Vax-11/780.

Pascal programs written for the PERQ range in overall speed from 1/5 to 1/3 the speed of comparable programs on the VAX 11/780, depending on whether 16-bit or 32-bit operations predominate. In fairness to the PERQ hardware, the underlying microengine is much faster than the Pascal timings in table 3-1 would indicate. Microcoded operations often run as fast as or faster than equivalent VAX 11/780 assembly language. Note the relative speeds of the microcoded context switch and kernel trap operations. Moreover, instruction sets better tuned to the PERQ hardware, such as the Accent CommonLisp instruction set, run at speeds closer to 50 percent of the VAX. Nevertheless, for the purpose of gauging the performance of the Accent kernel code, which is written in Pascal and makes heavy use of 32-bit arithmetic, pointer chasing and packed field accessing, the CPU speed of a PERQ is about 1/5 that of a VAX 11/780.

We used two different versions of Accent to make the measurements reported in this paper. Unless otherwise stated, all reported measurements were made on a PERQ I with a Shugart Winchester (24 megabyte, 85 millisecond average access) running a version of Accent (the ***profiling*** version) specifically modified to support microsecond timing. Instead of using statistical sampling, we revised the Pascal byte-code interpreter microcode to support procedure-level profiling by accumulating the time intervals between routine entry and exit. Both elapsed and process-virtual (CPU) time can be used as interval time bases with microsecond resolution. Both ordinary processes and the pager/scheduler can be profiled in both user and supervisor states.

Some measurements were made on a PERQ T2 with a MAXSTORE Winchester (140 megabyte, 30 millisecond average access) running a version of Accent (the ***standard*** version) more closely related

to the CMU standard distribution version of Accent. This version lacks the performance monitoring facilities of the profiling version, but includes improvements to the file accessing code including microcode support for operations on kernel data structures superceded by those discussed here.

3.2. The cost, frequency and distribution of IPC requests

Time	IPC Operation
1.15	Simple message send
1.35	Simple message receive
10.	Complex message send (1024 bytes)
10.	Complex message receive (1024 bytes)

Table 3-2: IPC operation times in milliseconds

Table 3-2 shows the costs of various forms of message passing in Accent. Simple messages are defined to be those with less than 960 bytes of in-line data that contain no pointers or port references (other than those in the message header). The times for complex messages were measured for messages containing one pointer to 1024 bytes of data. Simple messages are specially treated by the Accent kernel and thus are handled much more efficiently. The code for handling complex messages, on the other hand, has never been streamlined. During normal operation of the standard version of Accent, the observed ratio of simple to complex messages is approximately 12-to-1.[6]

Overall, the average number of messages per second observed during periods of heavy standard version use (e.g., compilation) is less than 30. 67378 simple messages and 4279 complex messages were sent during one measurement of three hours of editing, network file access, and text formatting, an average of less than eight per second.

3.3. The cost of virtual memory primitives

Five basic operations are at the heart of Accent virtual memory management:

1. *Virtual to physical address translation*

2. *ValidateMemory* creates new zero-filled virtual memory. Nearly all memory not mapped directly to Accent files is created using ValidateMemory.

[6] The system generation task described in section 3.5 shows a much smaller ratio of simple to complex messages, approximately 3.2-to-1. 46430 simple and 14489 complex messages were sent in 8043 seconds, about 5.8 simple and 1.8 complex per second. The increased density of complex messages is largely due to frequent file operations.

3. *InvalidateMemory* destroys virtual memory.

4. *MoveWords* transfers data by memory mapping as described in section 2.1.1. Most of the cost of copy-on-write data transfer is due to MoveWords.

5. *Page fault handling* also contributes heavily to the overall cost of virtual memory management. As described in section 2.2.5, it includes the cost of redeeming lazy evaluation: ZeroFill faults allocate physical caching while CopyOnWriteCopy operations do data copying.

3.3.1. The cost of address translation

Although Accent supports a fully paged virtual memory architecture, the PERQ itself has no address translation hardware. 32-bit virtual addresses are translated by microcode using the VP table described in section 2.2.4. The Pascal byte-code interpreter microcode maintains a reference-specific cache of virtual-to-physical address translations instead of a hardware address translation buffer. While the hit ratio for this cache is high, approximately 25000 to 45000 address translations through the VP table are still required per second of normal operation. A VP table address translation costs an average of about 6 microseconds. Approximately 15-30 percent of total runtime is thus spent doing address translations through the VP table. The cost of virtual memory in the absence of address translation hardware is at least that high, and also includes the costs of translation through the micro-caches.

3.3.2. The cost of validating and invalidating memory

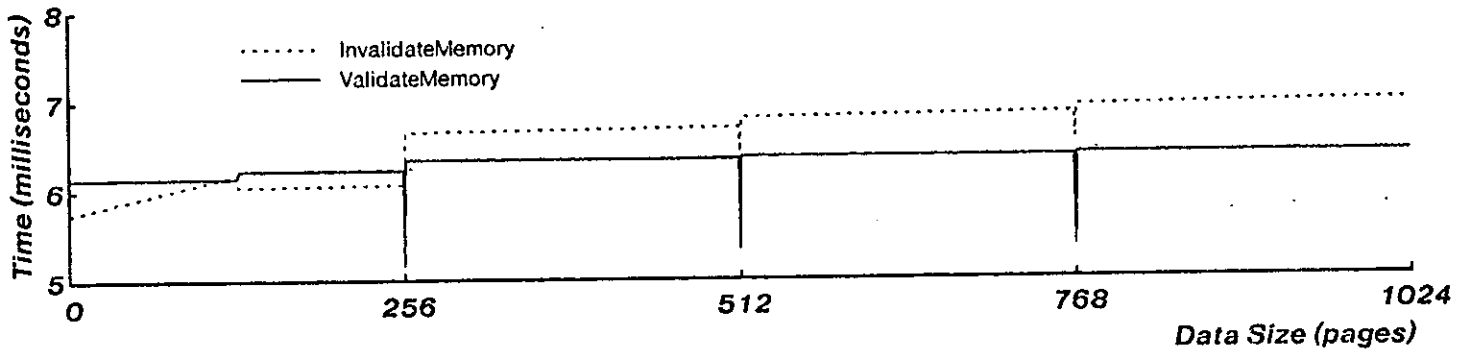

Figure 3-1: Validate/Invalidate of Unbacked Virtual Memory

To measure the cost of validating and invalidating memory, a test program repeatedly validated and invalidated from 1 to 1024 512-byte pages of data. The elapsed time for each operation was measured with a microsecond clock. Figure 3-1 shows the result of these measurements. Note that the base of the graph is at 5 milliseconds.

The most important observation about this graph is that the time cost of these operations is small

and increases very slowly with the size of the data. This reflects the design of the Accent process map and the lazy allocation of physical memory and backing store, neither of which are allocated during the ValidateMemory call. Allocation of physical memory is postponed until a process actually references the valid memory. Allocation of backing store is delayed until a dirty page of memory must be written out to disk.

The ValidateMemory operation therefore need only find an unused chunk of address space and mark it valid. The cost of this operation is nearly constant at 6.2 milliseconds. Several components of this total time can be identified. The ValidateMemory operation is provided by the kernel through a remote procedure call (RPC) message interface. The overhead of entering and leaving the kernel using the RPC interface is about 3.6 milliseconds. The cost of finding a hole in an address space is about 500 microseconds, with approximately 500 microseconds required to set up the level one and two process map tables. One millisecond is needed to allocate and set up the level three chunk descriptor.

The one millisecond downward spikes at multiples of 256 pages are due to the representation of unbacked memory in process maps. Chunks of 256 pages can be represented directly at level two in the process map without filling in the third level. Similar downward spikes at multiples of 256 pages also occur for other operations for the same or similar reasons. A related phenomenon occurs at multiples of 65536 pages, which can be represented at level one without either level two or three.

The InvalidateMemory operation removes a range of addresses from a process' address space. The time cost of InvalidateMemory on unbacked memory is higher than that of ValidateMemory because of the cost of searching the address range to reclaim any resources allocated to backing it. Although no such resources exist for unbacked memory, the kernel must still check for them. Information about disk backing is kept in the AST entries referenced by the process map. Caching in physical memory is recorded in the VP table. The cost of searching the process map is small and fairly constant because of the ability of the third level chunk descriptors to represent variable size regions of memory. About 700 microseconds are spent at levels one and two and approximately 550 microseconds at level three.

The cost of searching the VP table can be substantial. For small page ranges, the cost of searching by iterating up the address range adds little to the total cost of the InvalidateMemory call. Searching at about 3.5 microseconds per page produces the ramp at the left of figure 3-1. When the range is greater than 128 pages, the current algorithm for performing this search follows a chain of all VP entries belonging to the process. The cost of this algorithm depends on the length of the chain, i.e.

on the number of pages that the process has cached in physical memory. This chain is searched in microcode at a cost of about 3.5 microseconds per entry.

3.3.3. The cost of memory mapping with MoveWords

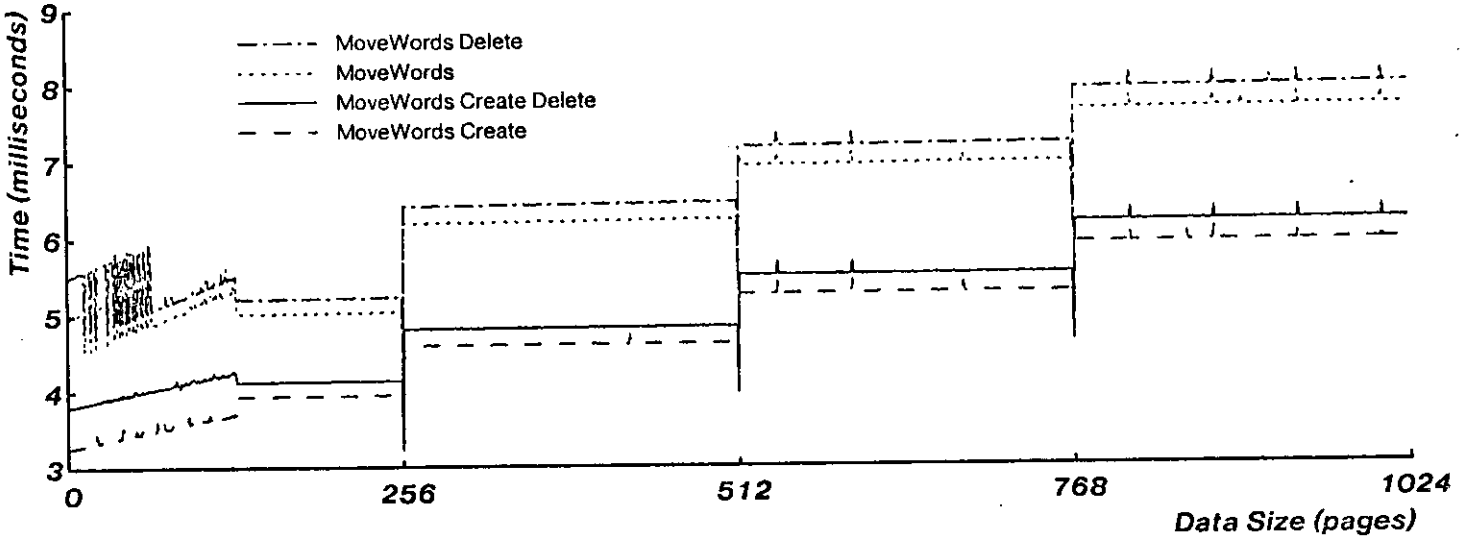

Figure 3-2: MoveWords of Unbacked Virtual Memory

Figure 3-2 illustrates the cost of mapping unbacked memory around with the MoveWords operation. Note that the base of the graph is at three milliseconds and that all operations show downward spikes at multiples of 256 pages and a ramp for sizes less than 128 pages.

The MoveWords operation is provided as a kernel trap, which costs 132 microseconds to enter and leave the kernel. The base cost of the MoveWords operation is thus somewhat lower than that of ValidateMemory and InvalidateMemory. Note that the cost within 256 page ranges is fairly constant, but that each additional 256 page chunk requires another entry at level two and level three, and costs slightly more (approximately 700 microseconds).

All MoveWords operations must scan the source address range (and the VP table) to transfer the specified mappings to the destination process map. The DeleteSource option can also be processed during this scan, so it costs only 200 microseconds for unbacked virtual memory, much less than the cost of another scan.

The MoveWords operations that create destination memory are cheaper than those that map into existing destination memory because the mapping operation needs to map into a hole in the destination address space. Creating new destination memory automatically guarantees such a hole. Mapping into a specified destination address range requires that the address range first be

invalidated to remove any existing memory. This costs another address range scan of the size of the data. Note, for example, that the MoveWords-Delete ramp is twice as steep as is the MoveWords-Create-Delete ramp.

Mapping into specified memory also has fragmentation problems. If the address range is not an exact multiple of the page size as it is in figure 3-2, the pages at one or both ends of the range will be partly outside of the range. In order to preserve any destination data on these partial pages, they must be copied rather than mapped.

Creating destination memory can circumvent these fragmentation problems by padding the data range out to the nearest page boundary and transferring the entire padded range by mapping. This gives the destination a small amount of undeserved source data, at worst a minor security leak that the source can plug by transferring only integral numbers of page-aligned pages.

The fragmentation problems of transferring into specific destination memory also increase the cost of the DeleteSource option. Because the source range to be deleted includes the partial end pages while the mapping operation excludes them, the DeleteSource cannot be combined with the mapping operation and requires a separate scan.

Because partially full end pages must be copied into specific destination memory, they cost more than unbacked pages. Reducing the data size to leave one half-full end page increases the cost of MoveWords from the 4.5-7.5 milliseconds shown in figure 3-2 to 10-14 milliseconds. The cost of MoveWords-Delete increases from 5-8 milliseconds to 13.5-17.5 milliseconds with a steeper ramp because of the third address range scan. The overall shape of the cost curve is otherwise preserved.

Figure 3-3 illustrates the cost of MoveWords on address ranges that are only cached in physical memory. The cost of manipulating the VP table representation of physical caching is dominant and produces curves that are roughly linear in the size of the data. The upper curve (Delete, with or without Create) has a slope of 1.75 milliseconds per page, the lower (no Delete, with or without Create) of 1.05 milliseconds per page. These costs primarily reflect the costs of entering and removing VP table entries. For each source page cached in physical memory, a new VP entry must be created to represent the destination access to the physical memory. Deleting source memory adds the cost of removing its VP entry. Invalidating destination memory as required when mapping into a specific destination address range also has the potential to remove VP entries and reclaim physical pages, although this never occurred in these tests.

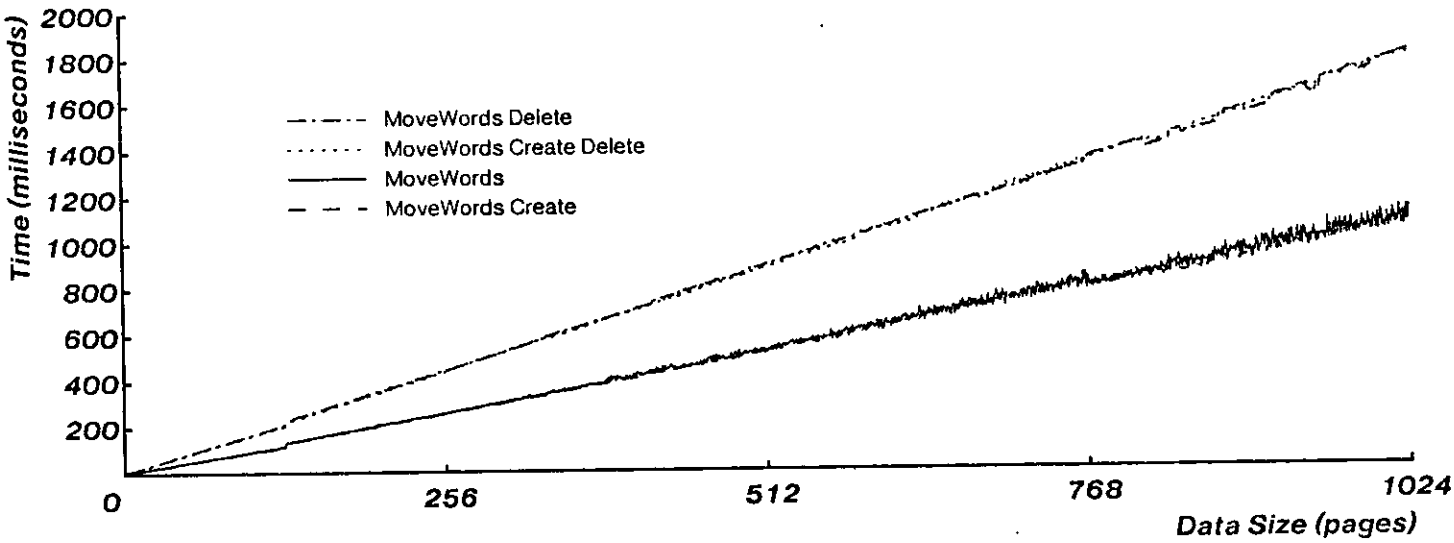

Figure 3-3: MoveWords of Physically Resident Virtual Memory

Since these measurements were made, the high cost of VP table handling led us to microcode the routines for entering and removing VP entries. The result was greater than a factor of ten reduction in the cost of the operations -- from about 800 microseconds to approximately 50 microseconds. The slope of the graph in figure 3-3 has thus been considerably reduced.

3.3.4. Fault handling

Dispatch	Map	AST	Page	VP	Total	Type of fault
.62					0.623	Null fault
.92	.43		.88	.88	3.355	Read fault, zero fill
1.28	.45		.91	.90	3.704	Write fault, zero fill
.92	.82	.48		.89	3.760	Read fault, memory fill, small file
.92	.60	1.49		.89	4.504	Read fault, memory fill, large file
.92			1.00	1.42	3.833	Write fault, CopyOnWrite copy

Table 3-3: Fault handling times in milliseconds
Fault times are about 3-4 milliseconds, not including any disk costs.

Table 3-3 summarizes the results from test programs that caused 100,000 instances of a variety of memory fault types. It shows the average *Total* times required to handle single faults and breaks these total times into components.

The *Dispatch* component is pure overhead. It includes about .16 milliseconds for two context

switches, about .44 milliseconds for demultiplexing the pager/scheduler and time to distinguish fault types from each other. It is therefore higher for more complex fault types.

The ***Map*** and ***AST*** components are the times required to look in the process map for the memory object, if any, backing the fault address and to find the correct page of that object. There is no memory object for zero-fill faults. The map lookup time is smaller for large files because fewer chunk descriptors each have more data in the map. The AST lookup time is larger because the code for accessing the two-level random indices used in large files has never been streamlined.

The ***Page*** component includes the cost of allocating a physical memory page from the free list, zeroing or copying data into it and recording its new use. If the free list were empty, the cost of the fault would increase by the cost of reclaiming a page, swapping it out first if necessary.

The ***VP*** component is the cost of entering a mapping for the fault virtual address in the VP table. For a CopyOnWriteCopy, it also includes the cost of removing the old mapping from the fault virtual address to the shared physical page.

3.4. The cost of mapped file access

All file access is mediated through the Accent memory management system. There are no separate file buffers maintained by the system or special operations required for file access versus access to other forms of process mapped memory.

System	Time	Operation
Accent	66	Request file from server
Accent	5-10	Read a page (512 bytes)
UNIX	5-10	Open/close
UNIX	16-18	Read a page (1024 bytes)

Table 3-4: File access times in milliseconds
Accent file reading performance is comparable to that of Unix4.1bsd.

Table 3-4 shows the costs associated with reading a 56K byte file under UNIX 4.1bsd on a VAX 11/780 with a 30 millisecond average access time Fujitsu disk and under the standard version of Accent with a 30 millisecond average access time MAXSTORE drive.

Handling the file request is clearly an expensive operation in Accent, even considering the

difference in speed between a PERQ and a VAX. This is due in part to the cost of a disk write to update the file access time. This disk write is unbuffered in Accent and thus is included in the file request time. The Unix disk write is buffered and is excluded from the open/close time.

Once mapped, file access in Accent ranges from somewhat faster than 4.1bsd to slightly slower, depending on the locality of file pages. 4.2bsd file access [22] is considerably faster than either 4.1bsd or Accent. This increase in speed appears to be due almost entirely to the larger (typically 4096 byte) file page size. The actual number of disk I/O operations per second under 4.2 is almost identical to 4.1, about 50-60 per second, and appears to be bounded by the rotational speed of the disk (60 revolutions per second).

Accent file access speed is limited by the basic fault time of about four milliseconds, the average number of consecutive file pages on a disk track and the cost of making new VP entries. Its page size is only 512 bytes, in contrast to 1024 bytes for 4.1bsd and 4096 or 8192 for 4.2bsd.

3.5. System generation task: distribution of system costs under load

The system generation task consists of a pair of command files that we routinely use to build the entire system from sources. This task provides a roughly repeatable test of computation and file intensive operation. This section considers the distribution of time costs during the test. Subsequent sections will consider the way virtual and physical memory are used.

The first command file retrieves about 6 megabytes of data from a file server in about 40 minutes. The other picks up about a minute after the first starts (when enough sources have reached the PERQ disk) and macro expands, compiles, links, and writes boot, runtime library and program files for the new system. It produces 1.8 megabytes of desired output and 2.9 megabytes of intermediate files in about 8040 seconds (about 2 hours 15 minutes).

Table 3-5 shows the breakdown of time in seconds spent by each process. For all processes except the pager/scheduler, ***User*** time is CPU time spent in user state, ***Sys*** time is CPU time spent in supervisor state by the kernel on behalf of that process, primarily performing IPC operations. Pager/scheduler ***Sys*** time includes time spent reading and writing memory objects (e.g. disk files), destroying process address spaces (see section 3.7.3), and handling page faults. All fault handling time is charged to the pager/scheduler, not to the process on whose behalf it is done. The otherwise unused pager/scheduler ***User*** time records microcode context switching time not charged to any process. It includes the costs of saving and restoring process microstate and low level process scheduling, about 40 microseconds per context switch and 70 microseconds per trap to and from

User	%	Sys	%	Count	Description
4089.455	50.8	140.510	1.7	248	pascal compiler
74.908	.9	1777.481	22.1	1	pager/scheduler
797.649	9.9	.000	.0	1	idle loop
420.758	5.2	9.268	.1	25	macro processor
149.655	1.9	182.816	2.3	1	file server
64.255	.8	40.248	.5	15	ethernet file transfer
78.545	1.0	9.227	.1	35	linker
4.531	.1	30.663	.4	1	process manager

Table 3-5: System generation task: cpu time in seconds by process
More than 61% of total time is delivered to ordinary processes in user state.

supervisor state. The column labeled ***Count*** shows the number of instances of that program that were executed. Each line shows the sum of the times charged to all count executions. The idle loop program runs at low priority to fill time during which no other process can execute, typically because the only processes trying to run are blocked in I/O waits, e.g. for the disk. A few other processes, notably user interface shells, account for the remaining small percentage of total time.

These measurements show that Accent has acceptable overall performance. More than 61 percent of total time is delivered to user processes in user state. 10 percent of total time is spent waiting for the disk. The less than 6 percent of total time spent in supervisor state by user processes is predominantly IPC cost. Of the 22.1 percent of total time spent by the pager/scheduler, 12.5 percent is spent handling page faults, 6.5 percent is spent doing operations (e.g. read or write) on permanent disk objects, and 2.3 percent is spent destroying process address spaces.

The two command files create/destroy 438 processes in this example, about 3.3 per minute. Extrapolation from the cost of the null program (350 milliseconds, see section 3.7) would indicate that roughly 2 percent of total time is spent creating/destroying processes and their server connections. The true cost is somewhat higher because processes with larger address spaces and more physical pages take longer to create and destroy. An average of 860 milliseconds (5 percent) should be closer to the mark.

3.6. Memory utilization

This section presents several measures of the use of virtual and physical memory by Accent. A breakdown of kernel code size by function indicates the complexity of various parts of the implementation. A static breakdown of physical memory use identifies the major consumers of physical memory. The dynamic working set of the system determines how much physical memory is left for user processes. The dynamic use of the paging pool shows how the system uses physical memory as a cache of virtual memory. The use of the MoveWords mapping primitive reflects the use of virtual memory by processes. Fault activity shows the effectiveness of lazy evaluation.

3.6.1. Kernel code size

Pages	KBytes	%	Kernel component
119	59.5	32.4	Paging and mapping
39	19.5	10.6	IPC facility
32	16.0	8.7	Memory object system
31	15.5	8.4	User process interface
4	2.0	1.1	Process creation/termination
142	71.0	38.7	Debugging and monitoring
367	183.5	100.0	Total

Table 3-6: Kernel code size: in 512-byte pages by function

Table 3-6 presents a rough breakdown by function of the total code size of the profiling kernel. It should be noted that the code density of PERQ Pascal is low compared to that of a VAX. Code sizes of typical PERQ Pascal programs are approximately 30 percent larger than their VAX C counterparts. Debugging code, such as print statements, is distributed throughout the body of Accent and thus the percentage of code devoted to diagnostics is actually larger than the 38.7 percent shown here. The code size of the CMU standard version of Accent, which also includes substantial debugging support, is approximately 119KBytes.

3.6.2. Static physical memory utilization

Table 3-7 identifies the primary consumers of physical memory. Accent locks down a minimum of memory: this implementation pages out of 73 percent of 2 megabytes with only 3 percent of locked kernel code. This is due both to movement of operating system functions into pagable server processes and to making much of the kernel itself pagable. Only 1/3 of the kernel code need be core-locked, for example.

Pages	KBytes	%	Description
192	96.0	4.7	Locked display buffer (768 x 1024 / (16 x 256))
14	7.0	.3	Misc. locked display memory
83	41.5	2.0	Other locked device memory
448	224.0	10.9	Locked VP Table (7 x 4 x 4096 / 256)
176	88.0	4.3	Locked PV Table (11 x 4096 / 256)
34	17.0	.8	Other locked kernel tables
41	20.5	1.0	Locked kernel stack, statics and other data
128	64.0	3.1	Locked kernel code
2980	1440.0	72.8	Paging cache memory pool
4096	2048.0	100.0	Total physical pages

Table 3-7: Static allocation of physical memory in 512-byte pages
Accent pages out of 73% of 2Mb of memory, and could use as much as 90%.

It also shows that kernel virtual memory tables account for more than half of the locked memory, an argument for a larger page size. The size computations shown for the VP (virtual to physical) and PV (physical to virtual) tables depend on the number of physical pages (4096). Quadrupling the page size to 2048 bytes would shrink the tables to 1/4 of their current size, adding 238 kilobytes (11.4%) more memory to the paging cache. Getting the display out of main memory would free up another 103 kilobytes (5%). Accent would then be paging out of almost 90 percent of 2 megabytes.

3.6.3. System working set size

Since Accent makes so much of the system code and data pagable, the operating system working set includes load on the paging cache as well the locked physical memory just described. The amount of memory locked or used while performing routine tasks, such as simple shell interactions, is an easy approximation of this size.

The profiling Accent with 2 megabytes of memory and a 768x1024 display locks down around 1120 pages and uses around 320 more, a total of around 720 kilobytes. The standard Accent with 2 megabytes of memory and a 1024x1280 display locks down around 1000 pages and touches around 600 more, a total of around 800 kilobytes. The standard Accent has a bigger screen but a smaller VP table with only two entries per physical page. It also has a much greater bulk of server and other miscellaneous code and data. Both systems leave more than 60 percent of memory for paging non-system processes.

3.6.4. System generation task: dynamic use of paging cache

Count	%	Description
84393	67.3	Allocated from free page list
36526	29.1	Reclaimed permanent file disk pages without process mapping
1876	1.5	Reclaimed dirty process pages with disk backing
1625	1.3	Reclaimed clean process pages with disk backing
742	.6	Reclaimed process pages without disk backing
284	.2	Reclaimed paging disk pages without process mapping
125446	100.0	Total physical page allocations

Table 3-8: System generation task: replacement of 512-byte pages
Most pages persist until they are explicitly deleted.

Statistics gathered by the physical page allocator and its LRU page replacement scan present a crude picture of Accent's dynamic use of the physical memory in its paging cache. Table 3-8 shows the breakdown of pages reclaimed to feed the physical page allocator during the system generation task. It indicates the distribution of pages that go unused long enough to become 'old'.

Deleting process access to pages without disk backing adds the pages to the free list. Deleting process access to disk-backed pages leaves processless disk pages. That 67.3 percent of the pages allocated came from the free list indicates the success of lazy backing store allocation. That 96.4 percent were free or processless indicates that ample physical memory was available for the task, since most pages remained resident until they were explicitly deleted.

The breakdown of pages examined by the LRU scan indicates the average content of the paging cache when the free list has been exhausted. 182130 of the 298028 pages scanned (60 percent) were disk pages not mapped to any process (the disk cache), only 115055 (40 percent) had process mappings. The size of the disk cache is another indication that ample memory was available for the task.

Section 2.2.6 described two constraints on page replacement caused by the single-level store design. Page replacement was never affected by the requirement that a minimal number of pages be free or mapped to disk. It was unable to allocate disk backing on 16295 of 41053 scans (39 percent), and had to pass over 843 pages that required disk allocation before being swapped out (5 percent of 16295).

Overall, Accent page replacement appears adequate, and could almost certainly be improved. It performed well on the system generation task, but was not stressed since the paging pool was of ample size for the task.

3.6.5. System generation task: use of MoveWords

*	Move Mb	%	*	Dupl Mb	%	*	Total Mb	%	Mb	Count	Operation
1	94.9	22.4	1	94.9	22.4	2	189.7	44.8	94.9	876	Process creation
3	202.5	47.8				3	202.5	47.8	67.5	3112	Reading files
1	10.4	2.5	2	20.8	4.9	3	31.2	7.4	10.4	2120	Writing files
			1	1.9	.5	1	1.9	.5	1.9	286	MoveWords traps
	306.3	72.4		117.0	27.6		423.3	100.0			Total MoveWords

Table 3-9: System generation task: MoveWords use in megabytes
Process creation and file reading account for 90% of MoveWords data.
70% of MoveWords data is deleted at the source as it is transferred.

Table 3-9 shows an approximate breakdown of the 423 megabytes tranferred by 14894 calls on the MoveWords mapping primitive during the system generation task of section 3.5. Each row accounts the MoveWords used by the ***Count*** uses of an ***Operation***, totalling ***Mb*** megabytes. The headings ***Move***, ***Dupl*** and ***Total*** designate MoveWords with and without DeleteSource and their sum. The heading '*' designates the number of MoveWords of that kind used by each operation. The heading '%' designates that entry's fraction of the 423 megabyte total MoveWords data.

For example, WriteProcessMemory, used twice in creating each of 438 processes, maps data once into the kernel (dupl) and once out of the kernel (move). Process creation accounted for 44.8 percent of the total data transferred with MoveWords during the system generation task. A total of 3138 files were read and 1006 written by user processes. Some of these were empty and are not included in the count. The file server also reads and writes directory files. These files are only read once and their contents cached in the file server address space, but they must be written each time a file is entered in or removed from a directory. User file data is mapped 3 times, kernel to file server to kernel to user, as described in section 2.1.4. Directory files are only mapped between the kernel and the file server. Most MoveWords traps are assumed not to DeleteSource.

Creating processes and mapping in files are the dominant users of MoveWords, combining for 92 percent of the total MoveWords data. These uses require few VP table operations because the

source process has not accessed most of the data even though, as we will see in section 3.6.6, much of it is physically resident in the disk page cache. Lazy creation of these VP table entries has paid off in decreased mapping costs.

VP entries are typically created and removed for data that is written to a file. These entries are often the only representation of the data's backing, due to the lazy allocation of backing store and lazy updates to process maps. This lazy evaluation precludes further lazy VP table manipulation.

The heavy use of the MoveWords DeleteSource option (72 percent of total MoveWords data) demonstrates the value of the transfer/invalidate composite. Even more of the mapping to write files could have used DeleteSource had the MatchMaker-generated interface not interfered.

The minimal use of MoveWords traps indicates Accent's thorough integration of memory mapping with inter-process communication. More than 99 percent of the data transfer with MoveWords occurred as part of message passing.

3.6.6. System generation task: fault activity

The design of Accent was based, in part, on the intuition that lazy evaluation would be effective in avoiding unnecessary data copying and allocation of backing store. This intuition is supported by fault statistics that show that much of the work postponed by lazy evaluation is never done at all.

Table 3-10 shows the breakdown of fault-related activities that occurred during the system generation task. Note that AddressFaults occur on 96.7 percent of faults while CopyOnWrite faults account for only 0.415 percent. CopyOnWrite reclaims account for 0.340 percent and CopyOnWrite copies only 0.074 percent of all faults. Reclaiming ahead in CopyOnWrite reclaim faults had little effect, each fault reclaimed an average of only 1.34 CopyOnWrite pages.

Lazy evaluation of data copying was extremely successful. Only 77 kilobytes (151 pages) of the more than 400 megabytes of data that were mapped CopyOnWrite were actually copied (.018%).

Lazy allocation of backing store was very effective. Of the 124362 total physical pages allocated, 40509 were backed on disk, most by pages of permanent files. Disk backing was lazy-allocated for the remaining 83853 physical pages (67%). Only 698 of these pages were ultimately written out to disk (.83%). Many them were used to back file server directory caches for the 16 directories created during the system generation task. The 218 (.26%) pages that were deallocated are a better indicator of the amount of temporary memory backed on disk.

201614	faults
116350	read faults
85264	write faults
194996	address faults
116148	from read fault entry
78848	from write fault entry
837	CopyOnWrite faults
686	CopyOnWrite reclaims of 917 pages
151	CopyOnWrite copies
124362	physical pages allocated
83702	filled with zeros
40509	filled from disk
151	filled with CopyOnWrite copies
83853	lazy allocations of disk backing
698	Paging partition disk pages allocated
218	Paging partition disk pages freed
150835	disk addresses looked up
110326	found in physical memory
40509	read from disk
298000	VP table entries created (and removed)
195000	created for fault addresses
41000	created for cached disk pages
62000	created by MoveWords

Table 3-10: System generation task: fault statistics

Caching of disk pages in physical memory worked well. 73 percent of the disk pages looked up were found cached in physical memory, only 27 percent had to be read from disk. The effectiveness of lazy backing store allocation and of disk page caching further confirms the ample supply of physical memory claimed by section 3.6.4.

VP table entries were created and destroyed during this test at an average cost of about 770 and 650 microseconds[7], combining for about 5.2 percent of total time. These operations not only comprise a time bottleneck in this implementation, their expense precludes more liberal use to reduce other costs.

79 percent of the VP entries were created during address fault handling, 65 percent because a process had faulted at that address and 14 percent to record disk pages cached in physical memory. Only 21 percent were created by MoveWords. This predominantly lazy creation of VP entries is appropriate, given the current expense of the operations. Less expensive operations could be used

[7] The cost of VPEnter and VPRemove varies in response to hash collisions in the VP table.

more freely, especially to reduce fault overhead costs by creating several entries with a single fault.

3.7. Analysis of kernel intensive operations: process creation and destruction

Detailed examination of the way Accent processes are created and destroyed offers several important insights into the workings of Accent and its use. These are interesting operations for several reasons:

- They are important operations because their cost and frequency make them significant contributors to overall system costs. Accent, like Multics [23] and Unix [30], typically executes each individual command in its own process. Section 3.5 attributes 3 to 5 percent of total time during the system generation task to process creation and destruction.

- They make heavy use of Accent's facilities. Process creation uses virtual memory mapping facilities to create an address space for the new process, to map program code and initialized data to their correct addresses, and to construct new stack and data areas. It also uses Accent's IPC facilities to establish communication channels to system servers. Program termination destroys the process' address space and communication state. More than 85% of total time is spent in the pager/scheduler or by other processes in supervisor state.

- They highlight several Accent design decisions, notably its emphasis on sharing and its movement of traditional operating system code out of the kernel into servers and other processes.

- They demonstrate that Accent's performance on kernel-intensive operations can be comparable to that of corresponding operations on more conventional systems. They also illustrate the effective use of Accent's facilities.

3.7.1. The structure of an Accent program

The layout of code and data in the address space of Pascal processes in the microsecond timer version of Accent was designed with two objectives: maximizing code sharing and minimizing process startup time. The runtime system, process initialization, server interface and general utility code are linked into a shareable library. Individual programs are linked relative to this library, and processes normally share a single copy of this library code at run time. In addition, program code and data are mapped into contiguous memory to reduce the number of mapping operations needed to build it. The runtime library and program file formats permit them to be mapped, copy-on-write, unchanged into a process address space.

Starting a child process requires two mapping operations, one to map the runtime library into the child address space and one to map in the program file. By write-protecting the code portion of the files in the parent address space, it is not necessary to explicitly write protect the code again in the child, as this protection is preserved by MoveWords. Initialized data is shared copy-on-write between

the executable file, the parent and the child. The desired sharing and protection falls out naturally from the normal Accent memory mapping and inter-process communication mechanisms.

Pages	Description	Pages	Description	Pages	Description
292	Pascal runtime library	71	Disassembler	15	Copy file
200	Pascal compiler	65	Telnet	13	Rename file
159	RPC stub compiler	49	Pascal macroprocessor	8	Delete file
88	Text editor	47	Shell	5	Type file
88	Microcode assembler	46	Pascal linker	2	Null program
82	Ethernet file transfer	26	Directory listing	2	Echo command line

Table 3-11: Sizes of executable files in 512-byte pages
The library is the greater part of the code size of most of the programs shown. This library can be shared even between processes running different programs.

Table 3-11 shows the relative sizes of the library and a collection of representative program files. File sizes are somewhat inflated by loader symbols, which occupy 5-15 percent of the files. Also recall that Pascal byte code is about 30 percent less dense than corresponding Vax code for C. The shareable runtime library clearly comprises the bulk of the code space of the programs shown. This library can be shared, even between processes running different programs, and requires no backing other than the permanent disk file.

3.7.2. The life cycle of an Accent process

The life cycle of an Accent process can be divided into three phases: *process creation*, *process runtime/running* and *process termination*. As a consequence of Accent's policy of moving traditional operating system code into user-state processes, the Accent kernel does not have a process initialization primitive such as the Unix exec(2). Instead, each phase is controlled, not by the kernel, but by a specific Accent process.

Process creation is controlled by the parent, normally a user interface shell. It creates a child process with an empty address space and uses WriteProcessMemory to initialize the child address space with code and data. It initializes the child process state, such as its program counter and stack pointer.

The parent shell also partially initializes the child communication state by giving the child access to server processes. New connections to the process manager and file server are required. Connecting to the display manager is unnecessary, since the parent's window is loaned to the child, but the

parent must reinitialize the window after the child exits. These costs are distributed between the parent shell and the servers. Server access in the form of port capabilities is transferred in an initialization message sent by the parent to the child.

Internal process initialization and runtime are controlled by the newly created child. Initialization includes processing the initialization message from its parent, initializing the runtime system and RPC interfaces to servers. Most of this work is done within the child, although it can involve communication with servers and implicitly includes fault handling by the pager/scheduler. The child then runs its main program. When the main program exits, the child process asks to be terminated.

Process termination, controlled by the Accent Process Manager, has three phases. First, the process manager destroys the IPC state of the process, producing messages notifying other processes of the child's death. Then the pager/scheduler process destroys the child address space. Finally, processes that had been communicating with the child react to notification of the demise of its ports. Server processes release any data structures they had created for it. The parent shell resumes if it had been waiting for the child to exit.

3.7.3. Null program task: distribution of costs

The following measurements of process creation and termination were derived three shell command files that repeatedly ran the null program (a program that initializes itself and then exits). The first shell command file logged the time difference across 10000 runs of the unprofiled null program. The second generated a breakdown of time spent in each Accent process and detailed profiles of server processes and of the pager/scheduler, also from 10000 runs of the unprofiled null program. The third command file profiled 10 runs of the null program itself.

The average time per null program execution was 345 milliseconds from the first command file and 357 milliseconds from the second, 3.5 percent slower. Microsecond profiling generally costs around 5 percent in overall speed, but the cost varies with the frequency of routine calls.

Table 3-12 shows a breakdown of the life cycle of the null program corresponding to the description in section 3.7.2. The elapsed times shown include the costs of remote operations and of handling page faults. The most important feature of this breakdown is its lack of major bottlenecks. Most operations could be made faster, but none is so slow that it dominates the overall performance.

Notable by their absence are the costs of initializing the child interfaces to the file system and display manager. The initialization of these, and potentially other, interfaces is lazy evaluated and is never done in this test. The child needs its interfaces to the kernel and process manager in order to

12	shell decides to run the null program
138	shell creates and partially initializes child
33	shell creates child process
8.4	shell creates empty process
18.7	shell validates child stack, code, data and bss areas
4.7	shell reads child process state
41	shell maps code into child address space
34	shell connects child to servers
15.4	shell connects child to file server
13.9	shell registers child with process manager
4.6	shell makes process manager the child's guardian
27	shell starts child
4.6	shell sets child process state
6.9	shell sends initialization message to child
4.9	shell resumes child
10.2	shell destroys its access to child server ports
90	child pages itself in and initializes itself
10.7	child decodes initialization message from shell
10.3	child initializes connection to kernel
14.0	child initializes connection with process manager
8.5	child initializes its runtime system
7.7	child calls its main program
98	process manager destroys child, sundry cleanup
39.9	process manager destroys child communication state
31.7	pager/scheduler destroys child address space and process
15.8	file system destroys child connection
5.0	process manager logs performance data
9	shell reinitializes its window

Table 3-12: Null program: elapsed milliseconds by operation

terminate itself, so these are actually initialized. Initialization of substantial chunks of the runtime system is also postponed permanently by lazy evaluation.

Fault handling by the pager/scheduler accounts for 53 milliseconds (15% of total). This time is distributed among the costs of the child initializing itself. Although the child's code pages are physically resident (due to prior executions of the program), VP entries must still be created for those pages that the child needs. New physical pages for its data must also be allocated and zeroed. Each execution of the unprofiled null program causes 15 faults, 9 DiskFill faults for code pages and 6 ZeroFill faults for data pages.

4. Related work

Virtual memory management and interprocess communication have traditionally been treated as independent subsystems. There are several reasons for this separation:

- Much of the early work on message based systems was done using small mini-computers (e.g., the PDP-11 [29], NOVA [7] and Eclipse [20]). These machines had limited virtual address space, which reduced the need for paged virtual memory.

- Message based systems have often been targeted to specific applications (e.g., real-time, process control [13]) which precluded or limited the applicability of operating system provided virtual memory support.

- Many early message systems were built on machines with primitive memory management hardware that made it difficult or impossible to build sophisticated virtual memory software. This was true both of the small minicomputer based systems and some large machines such as the Cray-1 [4]. Even early 16-bit microprocessors, such as the Motorola 68000, did not initially provide support for demand paging, making it difficult to build virtual memory support for such systems.

The notion of using memory mapping to provide access to shared memory in messages is not new: a number of traditional timesharing systems, among them Multics [23] and Tenex [6], have provided such facilities. The novelty of Accent is the degree of integration of virtual memory management and interprocess communication. The benefits of by-reference or P-MAP memory mapping are made available while preserving the advantages of by-value message passing semantics.

Apollo's Aegis [21] operating system shares with Accent the objective of permitting mapped access to data objects. Both systems view physical memory as a cache of virtual storage. Aegis, however, is built upon the use of shared read/write memory as its fundamental communication paradigm. Accent sidesteps the synchronization problems inherent in this approach through the use of by-value message passing for communication.

5. Conclusion

The successful use of Accent for a wide variety of distributed applications at CMU and elsewhere has shown that interprocess communication and virtual memory management can indeed be combined to form workable primitives for the design and implementation of a network operating system. Our measurements have demonstrated that these mechanisms can also be used to deliver single machine performance comparable to that of more traditional operating system designs. More than 61 percent of total time during a large-scale system generation task was delivered to processes in user state. The performance of kernel-intensive process creation and destruction operations is comparable to that of Unix4.1bsd on a VAX 11/780, after normalizing for differences in processor speed. Accent file reading performance is directly comparable to that of Unix4.1bsd.

Our measurements confirm that the cost of copy-on-write memory management is nearly identical to that of by-reference memory mapping. The overall contribution of copy-on-write faulting to total system costs is extremely small. Less than 0.01 percent of total time during the system generation task was spent handling such faults.

Lazy evaluation of memory map and backing store operations proved to be valuable. Of the physical pages for which backing store was lazy-allocated during the system generation task, fewer than 1 percent were ultimately recorded in process maps and backed on disk.

In Accent, unlike more traditional message systems, the cost of simple message passing is much less important than the cost of virtual memory operations. These costs are dramatically apparent in our measurements of the system generation task and of process creation and destruction. Ironically, far more care was taken in the Accent implementation to streamline simple IPC operations than to minimize virtual memory management costs.

The basic design of the Accent virtual memory system appears sound. Our measurements show that the costs of manipulating the Accent process map data structure to allocate, free and copy mapped regions are fundamentally small and grow slowly with the size of the affected memory area. The number of operations on the Virtual-to-Physical address translation (VP) table is limited by the number of physical memory pages owned by a process rather than the amount of virtual memory that it uses.

Unfortunately, the cost of actually taking a fault (3-4 milliseconds) or remapping a physical page of memory (800 microseconds in our original implementation) can easily dominate any process map manipulations. We have recently addressed these costs by moving the most expensive VP table

operations into microcode. Accent's use of microcode to speed up crucial operations is not unusual; architectural support for virtual memory and other abstractions is traditional. The PERQ is unusual in that it lacks address translation hardware but has a writable control store (WCS). Accent uses the flexibility of the WCS to overcome its speed deficiencies in the same way that other systems might use assembly language.

We have identified several other operations whose cost and frequency makes them performance bottlenecks and that are amenable to simple microcode, assembly language or hardware implementation. The operations of entering and removing VP table entries and iterating through the VP table are clear candidates. Specialized support for page replacement and searching process map tables would benefit the system substantially.

The effects of page size are important. Most system costs depend more strongly on the number of pages in a region than on the number of bytes in it. For largely historical reasons, Accent uses a 512-byte page. This small page size causes significantly more remapping of physical memory and more faulting operations than would occur with larger pages and reduces the effectiveness of address translation caches by reducing the size of the address range covered by a single cache entry. It also dramatically increases the costs of kernel data structures such as the VP table, whose size is a multiple of the number of physical pages in the system. The small disk page size often implies a large overhead to transfer a small amount of data. Experience with Unix systems [1, 22] indicates that the benefits of a larger page size would probably outweigh the costs of increased internal fragmentation.

Overall, the Accent implementation has satisfied its original goals. It provides an existence proof that a communication kernel with a few basic primitives can provide effective support for a large body of software. It has also demonstrated that a usable system can be built with its memory management and inter-process communication primitives and that these primitives can be implemented efficiently.

6. Acknowledgements

We would like to thank Gene Ball, George Robertson and Keith Lantz for their early contributions to the Accent design. Gene and George also labored heroically in helping to make the early Accent implementation a reality. Others contributed greatly to Accent's evolution: we particularly thank Doug Philips, Jeff Eppinger, David Golub, Mike Jones and Mary Thompson for their help. We would also like to thank David Cheriton, Jeff Eppinger and the TOCS referees for their comments on earlier drafts of this paper.

References

[1] Babaoglu, O. and W. Joy.
Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 78-86. ACM, December, 1981.

[2] Ball, J.E., E. Burke, I. Gertner, K.A. Lantz and R.F. Rashid.
Perspectives on Message-Based Distributed Computing.
In *Proc. 1979 Networking Symposium*, pages 46-51. IEEE, December, 1979.

[3] Barel, M.
PERQ Pascal Extensions.
Technical Report, Three Rivers Computer Corporation, June, 1979.

[4] Baskett, F., J.H. Howard and J.T. Montague.
Task Communication in DEMOS.
In *Proc. 6th Symposium on Operating Systems Principles*, pages 23-31. ACM, November, 1977.

[5] Birrell, A.D. and B.J. Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[6] Bobrow, D.G., J.D. Burchfiel, D.L. Murphy and R.S. Tomlinson.
TENEX, a Paged Time-Sharing System for the PDP-10.
Communications of the ACM 15(3):135-143, March, 1972.

[7] Cheriton, D.R., M.A. Malcolm, L.S. Melen and G.R. Sager.
Thoth, a Portable Real-Time Operating System.
Communications of the ACM 22(2):105-115, February, 1979.

[8] Cheriton, D.R. and W. Zwaenepoel.
The Distributed V Kernel and its Performance for Diskless Workstations.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 128-139. ACM, October, 1983.

[9] CMU Computer Science Department.
Proposal for a Joint Effort in Personal Scientific Computing.
Internal Document, Department of Computer Science, Carnegie-Mellon University, August, 1979.

[10] Corbato, F.J.
A Paging Experiment with the Multics System.
In Honor of P.M. Morse.
MIT Press, 1969, pages 217-228.

[11] Easton, M.C. and P.A. Franaszek.
Use Bit Scanning in Replacement Decisions.
IEEE Transactions on Computers c-28:133-141, February, 1979.

[12] French, R.E., R.W. Collins and L.W. Loen.
System/38 Machine Storage Management.
IBM System/38 Technical Developments, IBM General Systems Division :63-66, 1978.

[13] *GEC 4000 series computers: GEC4070, 4080, 4082 technical descriptions*
GEC Computers Limited, Elstree Way, Borehamwood, Hertfordshire, England, 1976.

[14] Hornig, D.A.
Automatic Partitioning and Scheduling on a Network of Personal Computers.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, November, 1984.

[15] Kahn, K.C. et al.
iMAX: A Multiprocessor Operating System for an Object-Based Computer.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 127-136. ACM, December, 1981.

[16] Jones, A.K., R.J. Chansler, I.E. Durham, K. Schwans and S. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proc. 7th Symposium on Operating Systems Principles*, pages 117-129. ACM, December, 1979.

[17] Jones, M.B., R.F. Rashid and M. Thompson.
Sesame: The Spice File System.
Internal Document, Department of Computer Science, Carnegie-Mellon University, October, 1982.

[18] Jones, M.B., R.F. Rashid and M. Thompson.
MatchMaker: An Interprocess Specification Language.
In *ACM Conference on Principles of Programming Languages*. ACM, January, 1985.

[19] Lampson, B.W. and D.D. Redell.
Experience with Processes and Monitors in Mesa.
Communications of the ACM 23(2):105-113, February, 1980.

[20] Lantz, K.A., K.D. Gradischnig, J.A. Feldman and R.F. Rashid.
Rochester's Intelligent Gateway.
Computer 15(10):54-68, October, 1982.

[21] Leach, P.L., P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson and B.L. Stumpf.
The Architecture of an Integrated Local Network.
IEEE Journal on Selected Areas in Communications SAC-1(5):842-857, November, 1983.

[22] McKusick, M.K., W.N. Joy, S.L. Leach and R.S. Fabry.
A Fast File System for UNIX.
ACM Transactions on Computer Systems 2(3):181-197 , August, 1984.

[23] Organick, E.I.
The Multics System: An Examination of Its Structure.
MIT Press, Cambridge, Mass., 1972.

[24] *PERQ System Overview*
Perq Systems Corporation, Pittsburgh, Pennsylvania, 1984.

[25] *Prime Reference Guide: System Architecture and Instructions*
Report IDR3060, Prime Computer, Inc., 1978.

[26] *Qnix System Reference Manual*
Perq Systems Corporation, Pittsburgh, Pennsylvania, 1985.

[27] Rashid, R.F. and G. Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 64-75. ACM, December, 1981.

[28] Redell, D.D. et al.
Pilot: An Operating System for a Personal Computer.
Communications of the ACM 23(2):81-91, February, 1980.

[29] Retz, D.L.
Elf: A system for network access.
In *Intercon Conference Record 25/2*, pages 1-5. IEEE, April, 1975.

[30] Ritchie, D.M. and K. Thompson.
The Unix Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[31] Spector, A.Z.
Multiprocessing Architectures for Local Computer Networks.
PhD thesis, Stanford, 1981.

[32] Spector, A.Z. et al.
Support for Distributed Transactions in the TABS Prototype.
In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, pages 186-206. October, 1984.

[33] Walden, D.C.
A System for Inter-Process Communication in a Resource-Sharing Computer Network.
Communications of the ACM 15(4):221-230, April, 1972.

[34] Wulf, W.A., R. Levin and S.P. Harbison.
Hydra/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

[35] Zayas, E.R.
Remote Paging in Accent.
1985.
in preparation.