NORTH-HOLLAND

An Implementation of an External Pager Interface on BSD UNIX

Hsiao-Hsi Wang, Pei-Ku Lu,[‡] and Ruei-Chuan Chang[§]
National Chiao Tung University, Hsinchu, Taiwan, and Academia Sinica, Naukang, Taipei, Republic of China.

We have designed and implemented an external pager facility on the ConvexOS for Convex supercomputers to enhance memory management capability within the kernel. A group of generic interfaces that can be used to construct a memory manager at user level is proposed. Furthermore, a highly modularized subsystem that provides dynamic configuration and portability is presented.

1. INTRODUCTION

The benefits of virtual memory go without saying: almost every high-performance computer in existence today has one. The UNIX operating system traditionally provides a memory manager embedded within the kernel. This manager is dedicated to solving all virtual memory-related problems, such as page faults, main memory shortage, and so on.

Because the clients of the memory manager include other components of the system, the manager requires some privileges and is therefore bound tightly within the UNIX kernel. As a result, user programs can participate neither in the decision making nor the paging strategy of the memory manager; only a single scheme for virtual memory management is provided through the entire operating system. However, in contemporary environments, one usually has specific requirements for backing storage management, consistency management, or paging mechanisms. It is obvious that a predetermined unique memory management policy within the kernel is not powerful enough.

Address correspondence to Ruei-Chuan Chang, Institute of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, Republic of China.

Mach (Tevanian, 1987; Young et al., 1987) is a multiprocessor operating system being developed at Carnegie Mellon University. The design of the virtual memory system separates the machine-independent portion from the machine-dependent one (Forin et al., 1989; Young et al., 1987; Young, 1989). Mach gives the user the ability to create memory objects, which can be managed by user-defined processes, called external pages.

In this article, we propose an external pager facility (Rozier et al., 1990; Tevanian, 1987; Young, 1989) on BSD UNIX to enhance the functionality of the memory manager. This facility enables users to construct their own memory manager outside the kernel in order to deal with user-specific requirements. Although the external pager feature originates from Mach, the design and implementation of our external pager have distinct advantages, described as follows:

- In the realization of our external pager in ConvexOS, which is a member of the BSD UNIX family, we divide the whole external pager subsystem into an operating system-dependent part, which is responsible for low-level architecture-related operations, and an operating system-independent part, which basically acts as the supervisor of the external pager subsystem. This applicability of the operating system-independent portion to any BSD UNIX operating system, which greatly simplifies the task of migrating the external pager facility, is not available with Mach. The consequence of adopting such a modularized implementation strategy is that the entire subsystem becomes highly portable and flexible.
- The implementation of our external pager retains the original system organization and just adds

0164-1212/95/$9.50
SSDI 0164-1212(94)00074-W

some hooks for the convenience of users. The default memory manager can still operate undisturbed within the kernel and cooperate with the external pager at any time.
- The similarity between our external pager facility and the Mach operating system makes the porting of relevant user application programs very easy.

The rest of the article is organized as follows. Section 2 provides a sketch of the architecture of the Convex supercomputer. Section 3 provides a detailed explanation of the external pager interface. In addition, the modularized implementation and an application of the external pager facility are described. Section 4 evaluates our work, and Section 5 summarizes some previous related work. Finally, concluding remarks and suggestions for future research are given in Section 6.

2. THE CONVEX MACHINE

2.1 A Close Look at the Convex Architecture

The Convex C1 and C2 machines manufactured by the Convex Computer Corporation adopt 4.2 BSD UNIX as their primary operating system and incorporate a number of enhancements on the original BSD UNIX implementation (CONVEX 1988, 1990, 1991).

The Convex C100 series architecture defines 4 Gbytes of virtual address space. From the point of view of hardware, this virtual memory is partitioned into eight 512-Mbyte segments. Logically, the 4 Gbytes of virtual address space of a processor are divided into five partitions, called rings. Each ring has a different level of privilege for execution and access, so the ring structure architecture embodies the fundamental memory protection mechanisms. The ConvexOS maps segments to rings as follows:

- Segment 0 is always assigned to ring 0, which contains primarily the operating system kernel instructions. Because a set of privileged instructions can be executed only in this ring, the kernel has the privilege to perform all these functions.
- Segment 1 is assigned to ring 1, which is used by the kernel to map page tables (described below) into virtual memory so that ConvexOS can operate on them through virtual addresses.
- Segment 2 is assigned to ring 2. To date, ring 2 has been left unused.
- Segment 3 is assigned to ring 3. Various buffers used to cache fickle kernel data structures are allocated here.
- Segments 4–7 are assigned to ring 4, which has the lowest level of privilege. This area is used for the user context of Convex process.

To govern the large virtual address space, the Convex architecture uses a sequence of data structures organized in a hierarchical fashion (Figure 1). The top level of management involves a set of eight segment descriptors. Each segment descriptor points to the beginning of a first-level page table. When a process is loaded for execution, the appropriate segment descriptors must be loaded into the CPU's segment descriptor registers (SDRs). The second and third stages of management are accomplished by use of a number of first- and second-level page tables. A page table is a page that contains 4-byte entries called page table entries (PTEs). Each PTE conveys information needed to determine whether a page is resident in physical memory or not. Figure 1 also shows a virtual-to-physical translation of a 32-bit virtual address by consulting the hierarchical tables.

2.2 Memory Management in ConvexOS

ConvexOS is a demand-paging virtual memory operating system for supercomputers. To provide a flexible and reliable virtual memory programming environment, four basic memory management data structures are used in ConvexOS (Figure 2):

1. Core map (cmap): an array of structures used to manage main memory.
2. Virtual space (vspace): A top level sketch of the virtual address space. The major component is composed of a linked list of entries termed *vm_region*.
3. Virtual memory object (*vm_object*): a single shareable unit that can be mapped into the virtual address space of one or more processes. Examples of objects are files, swap space (backing storage), and zero-filled pages.
4. Page tables: a machine-dependent memory-mapping data structure.

All knowledge concerning the virtual address space of a process is acquired through the use of the corresponding vspace data structure. Most of the information recorded in vspace is machine independent, for example, the current access mode of a process context, the permitted maximum resident size, and so on.

The only machine-dependent information in vspace is described in an array named *vs_sdr*. The contents of *vs_sdr* correspond to the eight SDRs required by hardware architecture and are loaded to

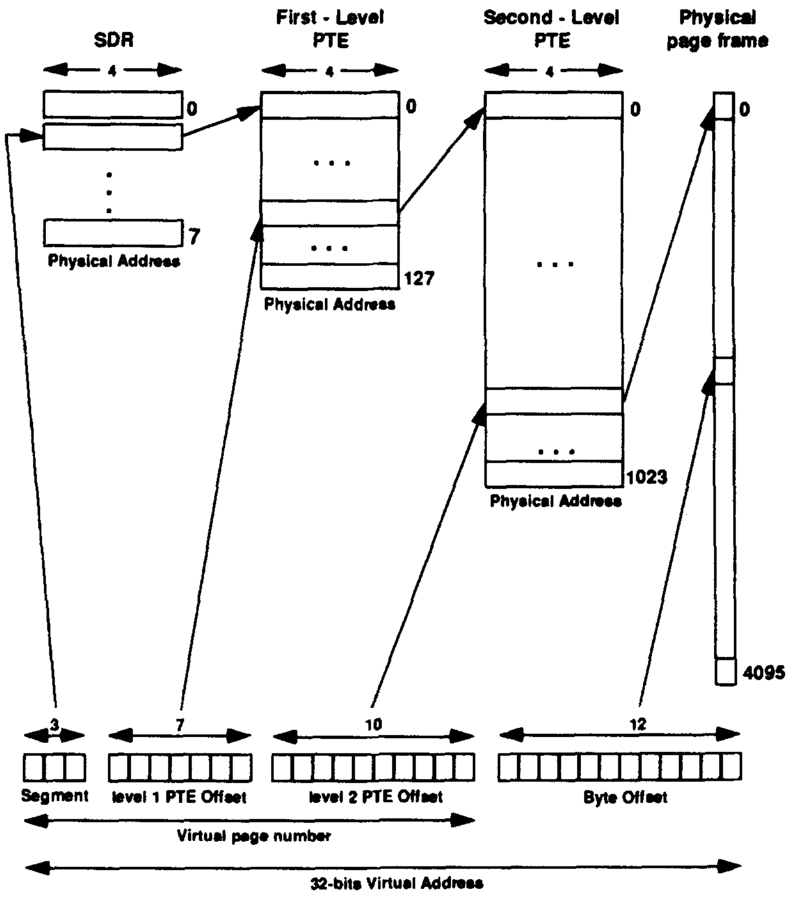


Figure 1. Virtual-to-physical address translation.

machine SDRs each time this process is scheduled for execution. When the SDRs are switched, the corresponding page tables are also exchanged, so the system views a different virtual address space.

From the point of view of a memory manager, the linked list indicated by the *vs_regions* field of vspace is the most important data structure. This linked list is sorted in order of ascending virtual address and

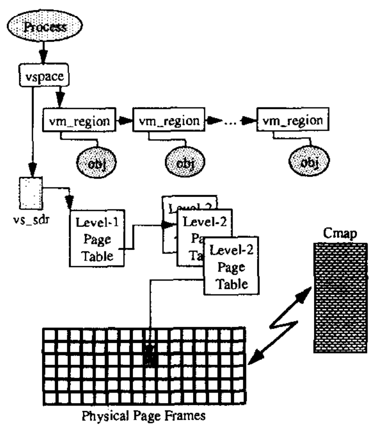


Figure 2. ConvexOS virtual memory system operating principles.

each entry (*vm_region*) maps to a contiguous range of virtual addresses. Each *vm_region* contains basic information such as the protection right specific to this region, the range of the region, and so on.

Basically, two *vm_objects* are attached to each region by fields in the region entry, called *re_object* and *re_backing*. The first (primary) object is used for providing pages to the page fault handler when the desired pages have not yet been paged out to backing storage, for instance, the zero-filled pages or the permanent text portion in execution programs. The second (backing) object is used to transfer swapped pages between main memory and backing storage.

The virtual memory subsystem of ConvexOS represents a typical form of the BSD UNIX. We use ConvexOS as a testbed to explore the interface required to support paging control activity outside of the kernel.

3. AN EXTERNAL PAGER FACILITY UNDER A BSD UNIX ENVIRONMENT

Normally, the UNIX operating system offers a memory manager embedded within the kernel. Such a memory manager performs all activities between backing storage and main memory (a page pool), i.e., main memory is used as the cache of backing storage. We refer to this job as cache management. User programs can neither see the memory manager nor participate in its activities. However, users might wish to write a memory manager if they have specific requirements for one of the following:

- Backing storage management (such as the source of storage or its format, e.g., compressed or encrypted).
- Consistency management of the backing store versus the memory cache. Examples include mapped file support, transaction-based virtual memory, and distributed shared memory.
- Paging mechanisms. The memory manager is directly involved with the paging mechanisms for the object it manages. It has indirect influence on overall paging policy by its actions.

To meet these requirements, a traditional operating system such as the BSD UNIX must be modified to provide users with the ability to write their own memory manager in user mode. This section focuses on the mechanism and implementation of such an external pager facility and describes an application that uses the external pager facility to construct virtual shared memory on distributed systems.

3.1 The Generic Interface Design

We have developed a prototype system of the external pager facility on ConvexOS by providing a group of interfaces among the user, kernel, and external pager. Because most of these interfaces are implemented as remote procedure calls, we use client and server models to explain these interfaces. The initiator of the request is the client; the one satisfying the request is the server. Application of this facility on distributed environments makes it possible for the user, kernel, and external pager to locate on different sites.

To provide an external pager facility, the kernel has to redirect the original processing flow for paging, forward the requests to the user-level manager, and then wait for a reply. In addition, the kernel must supply the external pager with the capability to access the contents of the pages it needs and to update related internal data structures, such as the cmap, the page tables, the *vm_regions*, and so on.

The operations described above can be divided into two parts, according to whether the initiator of the operation is the kernel or the manager. Basically, the kernel may launch the following types of requests to the manager:

- requests for data from the backing storage managed by the pager when a nonresident page fault occurs;
- requests for more access permission for the data cached in the kernel when a protection violation fault occurs (e.g., a write fault on a page previously marked as read-only);
- requests to flush modified cached data to the associated backing storage if the amount of free memory is less than a certain threshold.

On the other hand, an external pager may raise the following demands for cache management while performing consistency maintenance:

- require that the kernel writes the pages in question back to the manager;
- clear the cached data in main memory and mark the corresponding pages as nonresident;
- invalidate a range of the virtual address space of a certain process.

Based on these requests and demands, we construct a sequence of interfaces to implement an external pager. Table 1 summarizes the interface calls made by the kernel on the external memory

Table 1. Summary of the External Pager Interface (from Kernel to Pager)

data_init (backing storage to map, virtual address, process id)	Contacts the manager of a cache that is mapped for the first time, for initial handshake
data_terminate (backing storage, virtual address, size, process id)	Notification of removal from cache
data_request (backing storage, faulting virtual address, desired size, desired access, faulting process)	Request for a range of pages that the kernel does not have in its cache
data_unlock (backing storage, faulting virtual address, desired size, desired access, faulting process)	Requires more access permissions for a range of pages
data_write (backing storage, buffer, buffer size, offset in backing storage, process id)	Pageout of dirty pages from main memory
lock_completed (backing storage, virtual address, size, process id)	Completion of the requested paging operation

manager. We implement each call as a remote procedure call (RPC). That is, the function body lies on the external pager side. In an actual implementation, the arguments of these procedure calls are packed into a message and sent to the server for processing. Several fundamental items are included in this message:

- A backing storage, which indicates the memory manager and the destination of the message.
- A process identifier, which the manager may use to make cache management requests to the kernel. The identifier is somewhat different from the well-known UNIX process identifier.
- A virtual address combined with a size in bytes, which specifies the range of virtual address space to be operated on.

If a portion of the virtual address space is created to be used for the first time as the cache for a certain backing storage, and the backing storage is administered by an external manager, then the kernel makes a *data_init* call to the corresponding manager to establish a correlation between the kernel and the manager. If a backing storage is mapped into the address space of more than one process on different hosts (with independent UNIX kernels), then the manager receives an initialization call (*data_init*) from each kernel.

To process a cache miss (i.e., nonresident page fault), the kernel issues a *data_request* call specifying the range (usually a single page) and the current access desired. The manager is expected to return at least the specified data, with as much access as it can allow.

When a user process requests greater access to cached data than the memory manager has permitted (i.e., protection violation fault), the kernel issues a *data_unlock* call. The manager is expected to respond by changing the locking on that data when it is able to do so.

The memory manager may fix the use of cached data by the interface (described below) from the pager to the kernel. For example, it may wish to change the access right of cached data or ask the kernel to write back the modifications in question. In the latter case, the kernel uses the *data_write* call in response, just as when it initiates a cache replacement to squeeze main memory. In either case, the kernel calls *lock_completed* to indicate that it has finished the operation requested.

Finally, a *data_terminate* call is used to inform the memory manager that the kernel has completed its use of the given cache data, so the manager can destroy the data after performing appropriate bookkeeping.

Alternatively, the external memory manager can manipulate the cached data in the kernel through the interface functions listed in Table 2. Some of these functions are organized as UNIX system calls, and some are library routines for improving feasibility, but most of them are RPCs (function bodies residing inside the kernel). Unlike those summarized in Table 1, these RPCs are asynchronous calls, so they do not wait for a reply from the kernel.

A memory manager is a server process that responds to specific messages from the kernel in order to handle memory management functions for the kernel. To isolate the memory manager from the specifics of message formats, we provide two procedures, *recv_request* and *memory_server*, to handle a received message. The *recv_request* routine accepts the incoming messages and records the emitter of the received message in the argument client for future use. Then the external pager calls the *memory_server* routine, which parses the contents of the request, does all necessary argument handling, and invokes appropriate functions (Figure 3).

On reception of a *data_init* call from the kernel, the memory manager may respond with *data_ready* when it is ready to map the associating backing storage to the given virtual address.

The *data_supply* call is used to provide data with permission access right to the kernel when a page fault occurs. It is usually made in response to a *data_request* call made to the memory manager.

Table 2. Summary of the External Pager Interface (from Pager to Kernel)

recv_request (request, client)	Accepts paging request
memory_server (request, client)	Handles the request to call one of the interface functions
data_ready (client, virtual address, (size)	Confirms availability on completion of initial handshake
data_supply (client, virtual address, size, buffer, access right)	Provides page(s) data to the cache
data_unavailable (client, virtual address, size)	Data is unavailable, kernel has to zero-fill the pages
lock_request (client, virtual address, size, should_clean, should_flush, reply_to)	Cache control request, e.g., page flush
give_right (client, virtual address, size, access right, reply_to)	Grants more access permissions for the cache.

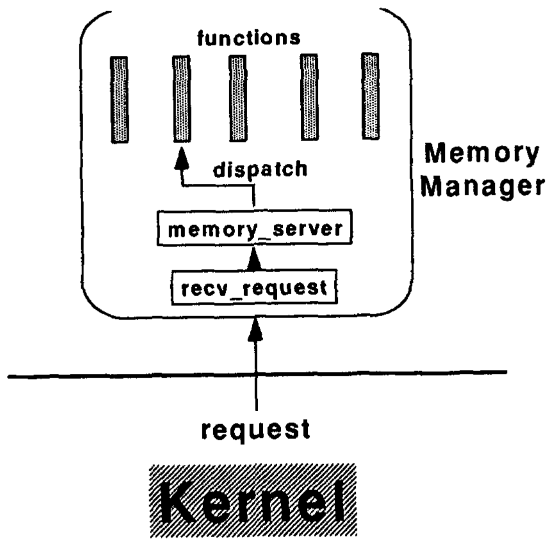


Figure 3. Dispatch of a kernel request.

However, the manager can still reply with a *data_unavailable* call to inform the kernel that the desired data are inaccessible.

To give greater access rights to cached data than previously granted, the manager may issue a *give_right* call to the kernel.

Because the memory manager may be subject to external constraints on the consistency of its backing storage, the interface provides a *lock_request* call to control caching. This call allows the memory manager to make the following requests of the kernel: write back any cached data that have been modified since the last time they were provided, and remove any uses of the data from memory (i.e., mark the data as nonresident).

A backing storage may be mapped into the address space of the application programs by exercising the *seg_alloc* primitive and specifying the memory manager responsible for the backing storage. The kernel redirects all the paging requests for pages in that virtual address space to the specified manager. However, this function call only designates unused pages in the bss region as cache for mapping; other segments, such as text or initialized data regions, that have been allocated before the program starts its execution cannot be remapped and are still managed by the default memory manager within the kernel.

An additional function, *seg_dealloc*, is proposed to be invoked explicitly by user programs to relinquish their access to a region previously declared by the *seg_alloc* call, so that further access to that memory fails. In addition, the operating system automatically deallocates the entire virtual address space of a process after quitting execution of that process. Table 3 lists these two virtual memory management functions.

Combining the primitives described above, we can construct an external pager to deal with the paging requests and administer the backing storage. Figure 4 shows the relationship between the kernel and the external pager.

3.2 Implementation of the External Pager in ConvexOS

We have altered the virtual memory subsystem of ConvexOS, a member of the BSD UNIX family, to provide an external pager capability. While establishing a set of reliable interface calls, we make full use of the original operating system, and no duplication of functions within the kernel is produced.

3.2.1 Some considerations. In developing the external pager facility, we want to achieve two major objectives: the potential for dynamic configuration and portability among various BSD UNIX operating systems.

Dynamic configuration removes the necessity for the pager to reside on the same host as the client process. Moreover, the relationship between the pager and a specific host is broken, and the pager can be freely reorganized without the intervention of the kernel or client processes. To attain this goal, the Berkeley socket interface (Stevens, 1990) is used as a basis for transactions.

The original operating system is modified to aid the mechanism of the external pager. To make the appended portion portable, the external page (XP) subsystem has to be modularized. In brief, it is divided into two logical parts, one that uses native functions to deal with such problems as operating system-dependent virtual memory administration, management of hardware-related PTEs, and so on, and another that is independent from the original operating system and thus can be easily migrated to other BSD UNIX systems. Figure 5 shows this organization.

Table 3. The Interface from the User to Kernel

seg_alloc (mapped size, returned virtual address, memory manager)	Maps a virtual memory region to the backing storage governed by the memory manager
seg_dealloc (virtual address, size)	Deallocates a virtual memory region

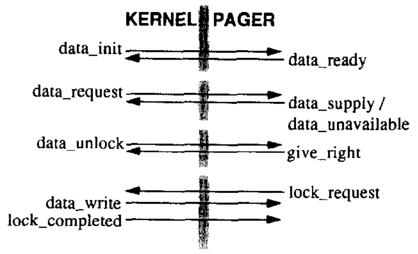


Figure 4. Interrelationship between the kernel and pager.

3.2.2 The prototype system. The XP subsystem consists primarily of four modules: segment administration, remote procedure invocation, request management, and cache control modules. The hierarchy of the system is shown in Figure 6.

Each segment is an integrated portion of memory cache served by a single external pager. The segment administrator implements the mapping between segments and the pager. Other than mapping, its main responsibility is to coordinate the default memory manager and the XP.

The administrator registers the server (the XP) of each allocated segment as a host and port number pair, just like the pairs used in the Berkeley socket. Because we currently work on an internet domain local area network, these pairs are quite enough to meet our requirements. However, for future needs, the registration for server should be expanded to include more information, such as domain family and protocol used.

Because the default memory manager still exists within the kernel and is obligated for most paging activities, there must be some way to distinguish the page faults for the normal cache from those for segments managed by an XP in order to send the paging requests to the proper manager. The Convex architecture hardware reserves four bits (bits 4–7) in each PTE for potential use by software (Figure 7). Two of these bits are taken up by ConvexOS: bit 4 indicates that the page data should be migrated to backing storage on flushing (otherwise it is discarded), and bit 5 denotes a page state of copy-on-write. Here, we adjust bit 6 as a flag to designate that the corresponding page is served by an XP. The segment administrator examines this flag to determine whether a paging request should be redirected; it thus coordinates the default memory manager and the XPs.

We have modified several routines in ConvexOS slightly to integrate them with the segment administrator. The *vm_grow* routine, which enlarges or shrinks the size of a region, must be able to mark the special status of the varied portion of the region (by setting the flags in the relevant PTEs). Two major routines for page fault handling in ConvexOS, *vm_zfod* for paging in nonresident pages of the bss region and *re_wtfault* for resolving protection violations, have also been modified to redirect requests to the segment administrator when necessary.

In brief, the segment administrator acts somewhat like a switching box or a filter, accepting requests coming either from the underlying operating system (i.e., for page faults) or from the user-level pager (i.e., for cache control), inspecting these requests and performing necessary bookkeeping, and then invoking proper routines for processing. It is the superior of the whole XP subsystem.

All the page faults are ultimately resolved by remote procedure calls to XP. Once the segment administrator completes the analysis of a paging request, the RPC module converts the procedure call to a full-rigged message and sends it to the

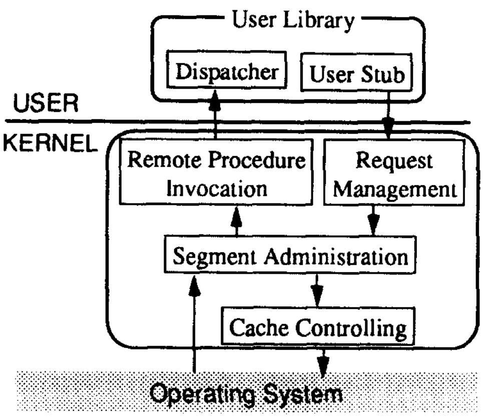


Figure 6. XP subsystem hierarchy.

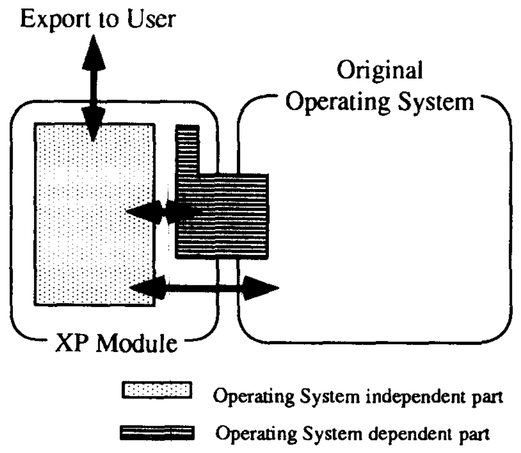

Export to User
Original Operating System
XP Module
Operating System independent part
Operating System dependent part

Figure 5. Organization of the XP subsystem.

V	Page Frame Base	HW	C	SW	RD	WR	EX	R
31	30 12	11 9	8	7 4	3	2	1	0

Figure 7. PTE format.

designated pager. Because a number of remote procedures may be called by the segment administrator, the procedure name is encoded in the outgoing message, and a user level dispatcher is introduced through the library to decode the received message.

The Berkeley socket interface is used as the basis of communication. However, the original socket interface is implemented as a system call and can only be used from the user level by a trap instruction. To use the interface from kernel mode, we have added a converter to reorganize the parameters for the socket and simulate the trap environment.

Another task carried out by the RPC module is process management. Because the faulting process and the XP are both user processes, the faulting process must be suspended after a request is sent; otherwise, the XP can never be scheduled for execution. The result is that the whole operating system is blocked in the kernel mode, and the faulting process is blocked continuously because it is waiting for a reply from the XP. The system is deadlocked. To prevent such a deadlock from occurring, the RPC stub exploits the socket interface to execute a context switch at an appropriate time.

The request manager handles the preprocessing of cache-controlling requests, which involves modifying page tables of the client process or accessing the cached data in the address space of the client process for the XP. These requests are always asynchronous to the execution of the client process. There exist two approaches to satisfy these requests. In the first approach, because the address space of each process is different and private, the XP has to map the page tables of the client process or the desired cache pages (which are in the context of another process) into an unused region in the address space of the XP in order to operate on them. This task is arduous because it requires extra locks to harmonize the XP with the client process. In the second approach, the request manager simply queues the request and sends a software interrupt to the client that requests the service; then the pager can immediately resume its execution. Once the client process is scheduled to run and it traps into kernel mode, the original request is then dequeued for examination, and the relevant routine is called to control the desired cache. This scheme reduces the complexity of the implementation through the elimination of verbose mapping. Therefore, we adopt the latter approach.

The cache control module provides all necessary low-level cache-controlling functions. It is the primary part of the XP subsystem. It is bound to the underlying operating system and machine hardware and thus must be entirely written when migrating the XP subsystem to other BSD UNIX operating systems.

From the viewpoint of the XP subsystem, the cache control module can be considered to be a set of functions provided by the original operating system. For ConvexOS, we define three functions: one to mark a range of virtual addresses as nonresident (i.e., for flush requests), one to copy the contents of physical page frames associated with particular PTEs into the buffer (i.e., for write-back requests), and the last to invalidate a range of virtual addresses (i.e., for segment deallocation requests).

Generally, these cache-controlling routines involve the modification of page tables. In ConvexOS, the level 1 and 2 page tables of currently running processes are mapped to the virtual address of segment 1, so the kernel can access the page table entries directly through the virtual address. We use this characteristic to efficiently program the cache-controlling routines.

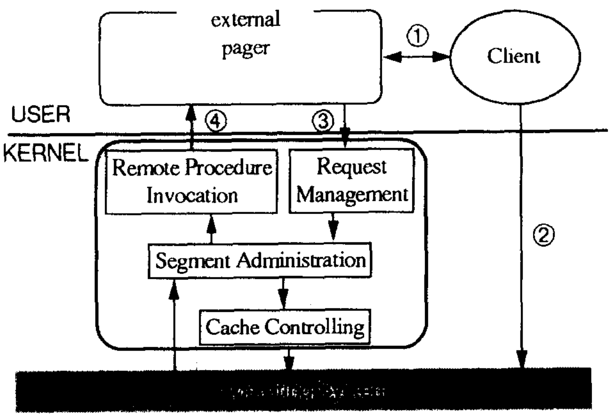
After introducing interface calls and implementation details, we use an example to explain how these interfaces locate in the XP subsystem and the paging transaction flow. For example, if a client process wants to map a specific backing object into its address space, it first uses socket interfaces to contact the XP to get an identifier (generally a UNIX socket port) and then pass this identifier as a memory manager to the kernel through the *seg_alloc* call. When page faults occur, *vm_zfod* or *vm_wtfault* is involved. These routines redirect faulting requests to the segment administrator. The segment administrator determines which manager (default pager or XP) will manage the request and then forwards paging requests to this manager (if this manager is an XP, then the interface from the kernel to the pager is applied through the remote procedure invocation module). Alternatively, the manager may also use the interface from the pager to the kernel to emit some cache-controlling requests through the request management module to the segment administrator, and the cache control module satisfies these requests. In the termination phase, the client process sends a *seg_dealloc* call to the kernel, then the kernel uses the *data_terminate* interface to inform the memory manager. Figure 8 illustrates this transaction flow.

3.3 An Application of the XP Facility

A distributed shared-memory (DSM) system provides the abstraction of shared address space among the computers connected by a network. This abstraction simplifies the sharing of complex data structures and the effort of writing parallel programs for these computers. However, the performance of a DSM system is highly dependent on how to make the shared memory consistent (Li and Hudak, 1989) and how to reduce the high cost of moving data through networks.

For ordinary shared data, we can exploit the interface between the XP and the kernel to support virtual shared memory and consistency management on distributed systems. Each time a page fault occurs, the kernel can catch this trap and use interfaces supported by the XP to send requests to the XP of this page. The XP can then guarantee consistency among various hosts.

Synchronization objects sometime result in heavy network traffic if we treat them as ordinary shared data. A set of library routines can be provided in a DSM system. These may include *mutex_lock*( ), *mutex_unlock*( ), and *barrier_wait*( ). We believe that implementing them by use of a synchronization server can prevent frequent movement and wasteful space of shared memory.



①Client uses socket interfaces to contact with pager to get port.
②The interface from the client to kernel.
③The interface from the pager to kernel.
④The interface from the kernel to pager.

Figure 8. Interaction among the client, kernel, and external pager.

4. DISCUSSION

4.1 Portability

After enhancing ConvexOS by adding the XP facility, the amount of code for the kernel increases by 16 Kbytes, 12 of which belong to the text region; the other 4 Kbytes belong to the data region. These sizes are negligible, especially when compared with the original ConvexOS, which exceeds 1.5 Mbytes of code. In fact, these 16 Kbytes of code need to be rewritten when porting the XP subsystem on a typical BSD UNIX operating system.

We divide the expanded code for the XP facility into two parts. The operating system-independent part occupies almost 50% of the code (Figure 9); most of this code is used for bookkeeping tasks. This part has no relationship to any operating system and can be migrated to other systems without modification. The operating system-dependent part can be further partitioned into two portions. One portion is related to the operating system but is common to all BSD UNIX systems. Examples of these codes include the socket interface converter in the RPC stub, the system call handler, and so on. When migrating the XP subsystem to other systems, these codes require little or no change. The other portion is the ConvexOS-specific portion, which uses individual features of ConvexOS, such as page table manipulation, and therefore has to be rewritten completely. The code in this portion accounts for only one fifth of the entire XP subsystem.

In summary, 80% of the expanded code is independent of the underlying operating system. This fact, in addition to the modularized implementation of the XP subsystem, makes porting the XP functions to the BSD UNIX family a simple task.

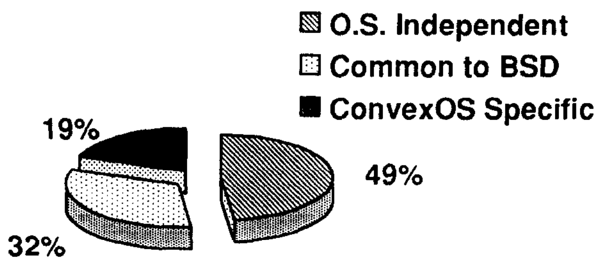


Figure 9. Relative portions of the XP subsystem code.

4.2 Dynamic Configuration

In implementing an XP interface, applicability for distributed environments is a major consideration. We have used the Berkeley socket as the basis for communication in our XP subsystem in order to extend the service scope of the XP to the entire network. Two of the sets of the proposed interface, the kernel to pager and pager to kernel interfaces, are implemented as RPCs if possible. The details about the cost of some operations are shown in Table 4. Although some overhead (an average of ~ 2.2 msec per page fault, which is comparable to 4.5 msec per null remote procedure call) is introduced by the native socket interface because of the complex assembly and disassembly processing of the network packet, it is still worthwhile. First, the XP provides a typical base for users to build their own memory managers. Second, the potential for dynamic configuration improves the feasibility of the XP interface for a distributed system. In fact, when compared with the network propagation delay, the overhead involved in using a Berkeley socket is negligible.

To further evaluate performance of the XP subsystem, we have implemented an external pager to mimic a default pager for managing paging behavior. A number of application programs, which include successive overrelaxation (SOR), Livermore loops, and n-body problem, are run under both default pagers and XPs.

The SOR problem is one of the most important computations arising in engineering and scientific applications, e.g., digital signal processing. The problem can be described as follows. Given a grid area represented by a matrix, each matrix element corresponds to a grid point. During each iteration, each matrix element is updated to average four neighbor points, which can be formulated as follows:

```
for ($k = 1$; $k \Leftarrow$ iterations; $k++$)
for ($i = 2$; $i \Leftarrow size - 1$, $i++$)
for ($j = 2$; $j \Leftarrow size - 1$; $j++$)
$A[i][j] = w * A[i][j] + (1 - w) * \frac{1}{4} * Sum\ of\ (4\ neighbors)$
```

where *size* is the size of matrix A, *iteration* is the number of iterations, and w is a constant.

We now turn to the problem of Livermore loops, which contain an initialization loop, loop 1, and loop 7. The initialization loop, which initializes matrix U, has only one nested DO-loop. The other two loops have two nested DO-loops, which have the following form:

```
for ($i = 0$; $i < size$; $i++$)
for ($j = 0$; $j < size/2$, $j++$)
$U[0][j] = func(U[1][j], U[2][j], U[3][j])$
```

where *size* is the size of matrix U and *func* is a numeric function.

The last problem considered is the n-body problem, which is one of the most famous problem in celestial mechanics. Suppose there are n homogeneous bodies in spherical layers; then they will attract each other as though their masses were at their centers. Let $m_1, m_2, \ldots, m_n$ represent their masses. Let the coordinates of m_i referred to a fixed system of axes be x_i, y_i, z_i ($i = 1, 2, \ldots, n$). Let $r_{i,j}$ represent the distance between m_i and m_j. Let k^2 represent a constant depending on the units used. Then the components of force on m_1 parallel to the x axis are

$$-\frac{k^2 m_1 m_2}{r_{1,2}^2} \cdot \frac{(x_1 - x_2)}{r_{1,2}}, \ldots, -\frac{k^2 m_1 m_n}{r_{1,n}^2} \cdot \frac{(x_1 - x_n)}{r_{1,n}}$$

and the total force is their sum. Then the components of force on m_1 parallel to the y axis and z axis can be computed similarly.

In our experiments, the problem sizes are 128 × 128 for SOR, 4096 × 4 for Livermore loops, and 2,048 bodies for the n-body problem, respectively. Detailed execution times and performance differences of all application programs are shown in Table 5. In summary, our experiments have shown that managing paging activities with an XP requires little overhead. The average performance degradation of these applications is 0.11, 0.73, and 0.40%, respectively. Most of these degradations come from the overhead of RPC interfaces between the kernel and

Table 4. The Cost of Some Operations

Operation	Elapsed Time (msec)
System call (getpid( ))	0.1232
Null remote procedure call	4.5365
Page fault (default pager)	3.3122
Page fault (external pager)	5.5180

Table 5. Run Time of Applications with Default Pager and XP

Application Programs (size)	With Default Pager (seconds)	With XP (seconds)	Difference Seconds	Difference Percent
SOR (128 × 128)	31.378245	31.415488	0.037243	0.11
Livermore (4096)	229.551529	230.457477	0.905948	0.40
N-body (2048)	117.858674	118.713899	0.855225	0.73

the XP. When compared with total run time, this overhead is negligible.

4.3 Overhead

To assess the extra cost introduced by the XP subsystem into the original operating system, we developed a program that allocates memory in the bss region and accesses data for each page. The running times of this program on the original ConvexOS and the enhanced ConvexOS were then compared. Figure 10 summarizes the results of our experiment.

The measurements show that the overhead due to the XP subsystem is nearly zero. Only in the case of a very large program with up to 64 Mbytes of virtual space was there a slight difference in the elapsed time needed to run the programs in the two systems (1% overhead, 0.038 seconds). This difference is vanishingly small when compared with the total run time, however, and such large programs are rarely used.

5. RELATED WORK

To application programmers, perhaps the most attractive feature of an XP interface is that it supports backing storage management (Loepere and Black, 1992) and, in particular, enables the behavior of kernel page in/page out activities to be controlled. Extracting the memory manager from the kernel has been the focus of much research in recent years.

Mach (Rashid et al. 1987; Tevanian, 1987) provides a relatively rich set of virtual memory management functions compared with systems such as 4.3 bsd UNIX or System V. Mach further complements these functions with its interprocess communication capabilities (Young et al., 1987) to efficiently transfer large regions of virtual memory in memory between protected address space for different processes.

The Mach XP (Young, 1989) interface consists of two parts. System calls are provided that allow user code to control the contents of the physical memory cache that contains the pages used by the client address space. The user also supplies a set of routines that are called by the kernel via an RPC-like interface, i.e., the MIG-generated interface, to handle such items as page faults and memory protection violations. Applications can freely use these virtual memory management interfaces to construct their own memory managers at user level and thus even define new paging rules, such as how to maintain consistency between the page images held by the kernel in its physical memory cache and the page images a manager might hold. An example of work on the Mach pager interface is DSM (Chang, 1991; Forin et al., 1989).

The Chorus/MIX operating system (Rozier et al., 1990), a formal research project on distributed systems at INRIA, demonstrates the feasibility of a UNIX implementation with a minimal kernel. Like the Mach design, Chorus provides a cluster of interfaces, called the generic memory management interface (GMI; Abrossimov 1989a, 1989b).

Through the mechanisms GMI supports, a memory manager (a segment mapper in Chorus) can reside completely outside of the kernel and simultaneously maintain memory segment consistency if the

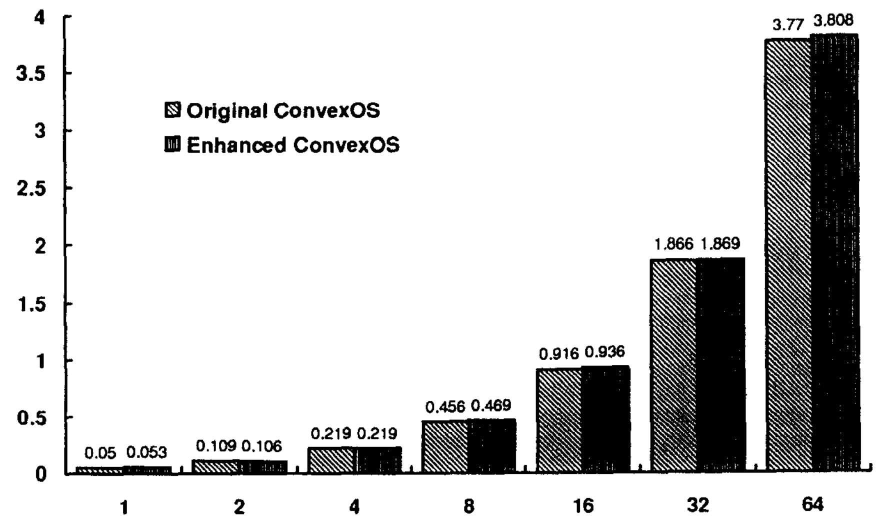


Figure 10. Memory allocation overhead for modified ConvexOS.

segment is shared among different sites. In particular, this interface provides abstractions for the support of a single consistent cache for both mapped objects and explicit I/O and the control of data caching in virtual memory. Data management policies are delegated to external managers.

The externality of the memory managers in Mach and Chorus is due to the nature of their kernelization. For the sake of a modularized minimal kernel, it is essential to separate each subsystem of a traditional monolithic operating system and clearly define the interfaces and negotiation protocols among each subsystem. In most cases, the internal implementation of structure is considerably modified to expedite this goal. For example, the Mach virtual memory subsystem is modified to make full use of its IPC module to improve overall performance.

In contrast, the implementation of our XP retains the original system organization and just adds some hooks for the convenience of users. The default memory manager can still operate undisturbed within the kernel and cooperate with the XP at any time. Moreover, we divide the whole XP subsystem into two parts: an operating system-dependent and an operating system-independent portion. This applicability of the operating system-independent portion to any BSD UNIX operating system, which greatly simplifies the task of migrating the XP, is not available with Mach or Chorus. Finally, the similarities between our XP facility and the other two systems make the porting of relevant user application programs very easy.

There exists much literature exploring the extended interfaces of the XP facility. For example, PREMO (McNamee and Armstrong, 1990) supports user level page replacement policies; Subramanian (1991) describes a Mach XP to manage discardable pages; Harty and Cheriton (1992) use external page-cache management to provide applications control of physical memory.

6. CONCLUSION

In a distributed computing environment, the diverse requirements of different types of applications highlight the inadequacy of uniform memory management policy inside the kernel. This article proposes a set of generic interfaces that allow users to design their own paging policy for memory manag 'nent. A modularized implementation structure is also presented to minimize the dependence of the XP on the underlying operating system, thus making the whole function portable.

By providing such a highly portable XP, we have shown that moving part of the task of memory management, especially backing storage administration, outside the BSD UNIX kernel is not only possible, but also beneficial. This is a significant departure not only from specific operating systems such as ConvexOS, but also from traditional UNIX operating systems, in which paging activities have been an integral part of the kernel.

A number of experiments on our XP have been made or are now in progress. An XP applying Li and Hudak's (1989) algorithm to maintain consistency between the memory cache and backing storage is currently running on a Convex supercomputer; implementations of mathematical applications such as matrix multiplication and inner product are also in progress, and their paging behaviors are being examined.

There are considerable opportunities for future work in this area. Moreover, our work provides a foundation on which related research can be based. Here we list a few suggestions for future research:

- The transaction between the kernel and the XP is now based on the Berkeley socket interface. Though this interface provides convenience in implementing an XP, it also introduces communication overhead. An optimization process is needed for minimizing overhead.
- To provide a sufficiently generic and flexible interface for XPs, application programs should be analyzed to investigate paging behavior.
- Currently, only part of the task of memory management can be transferred to the user level pager. Exploration of the minimum set of interfaces (or privileges) necessary to move the entire memory manager out of the kernel of a traditional UNIX operating system is an interesting topic.

ACKNOWLEDGMENTS

This research was partially supported by the National Science Council of the Republic of China under grant No. NSC81-0408-E009-18 and by the Convex Computer Corporation.

REFERENCES

Abrossimov, V., Rozier, M., and Shapiro, M., Generic virtual memory management for operating system kernels, in *Proceedings of 12th ACM Symposium on Operating Systems Principles*, 1989a, pp. 123-136.

Abrossimov, V., Rozier, M., and Gien, M., Virtual memory management in CHORUS, in *Proceedings of Progress in Distributed Operating Systems and Distributed Systems Management*, Springer-Verlag, Berlin, 1989b.

Chang, S. G., The Design and Implementation of Distributed Virtual Shared Memory, Master's Thesis, National Chiao Tung University, Taiwan, R.O.C., 1991.

CONVEX, *CONVEX Processor Operation Guide*, CONVEX Computer Cooperation, 1988.

CONVEX, *CONVEX Architecture Reference*, CONVEX Computer Cooperation, 1990.

CONVEX, *Managing ConvexOS: Configuration*, CONVEX Computer Cooperation, 1991.

Forin, A., Barrera, J., Young, M., and Rashid, R., Design, implementation, and performance evaluation of a distributed shared memory server for Mach, in *Proceedings of 1988 Winter USENIX Conference*, 1989, pp. 228–243.

Harty, K., and Cheriton, D. R., Application-controlled physical memory used external page-cache management, in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 187–197.

Li, K., and Hudak, P., Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. Comp. Syst.* 7, 320–359 (1989).

Loepere, K., and Black, D. L., *Mach 3 Server Writer's Guide*, Open Software Foundation and Carnegie-Mellon University, 1992.

McNamee, D., and Armstrong, K., Extending the Mach External Pager Interface to Allow User-Level Page Replacement Policies, Technical Report 90-09-05, University of Washington, 1990.

Rashid, R., et al., Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures, in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987, pp. 31–39.

Rozier, M., et al., Overview of the CHORUS Distributed Operating Systems, Chorus System Technical Report CS/TR-90-25, Chorus systèms, 1990.

Stevens, W. R., *UNIX Network Programming*, Prentice-Hall, 1990.

Subramanian, I., Managing discardable pages with an external pager, in *Proceedings of the Second USENIX Mach Symposium*, 1991.

Tevanian, A., Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments, Ph.D. Thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1987.

Young, M., Exporting a User Interface to Memory Management from a Communication-Oriented Operating System, Ph.D. Thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1989.

Young, M., et al., The duality of memory and communication in the implementation of a multiprocessor operating system, in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, 1987, pp. 63–76.