Design and Implementation of the Berkeley Virtual Memory Extensions to the UNIX† Operating System‡

Özalp Babaoğlu

William Joy

Juan Porcar

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This paper describes a modified version of the UNIX operating system that supports virtual memory through demand paging. The particular implementation being described here is specific to the VAX*-11/780 computer system although most of the design decisions have wider applicability.

The modified system creates a large virtual address space for user programs while supporting the same user level interface as UNIX. The few new system calls that have been introduced are primarily aimed for performance enhancement. The paging system implements a variant of the global CLOCK replacement policy (an approximation of the global *least recently used* algorithm) with a working-set-like mechanism for the control of multiprogramming level.

Measurement results indicate that the lack of *reference bits* in the VAX memory-management hardware can be overcome at relatively little expense through software detection. Also included are measurement results comparing the virtual system performance to the swap-based system performance under a script-driven load.

Keywords and phrases: UNIX, virtual memory, paging, swapping, operating systems, performance evaluation, VAX.

18 May 2004

† UNIX and UNIX/32V are Trademarks of Bell Laboratories

‡ Work supported by the National Science Foundation under grants MCS 7807291, MCS 7824618, MCS 7407644-A03 and by an IBM Graduate Fellowship to the second author.

* VAX and PDP are trademarks of Digital Equipment Corporation.

Design and Implementation of the Berkeley Virtual Memory Extensions to the UNIX† Operating System‡

Özalp Babaoğlu

William Joy

Juan Porcar

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

1. Introduction

The most significant architectural enhancement that the VAX-11/780 provides over its predecessor, the PDP*-11, is the very large address space made available to user programs. The fundamental task of transporting UNIX to this new hardware was accomplished by Bell Laboratories at Holmdel. In addition to the portability directed changes, the memory-management mechanism of the base system was modified to make partial use of the new hardware. In particular, through these changes, processes could be *scatter loaded* into memory thus avoiding main-memory fragmentation, and swapped in and out of memory *partially*. A process, however, still had to be fully loaded in order to execute. While no longer limited by the 16 bit address space of the PDP-11, the per-process address space could grow only as large as the physical memory available to user processes. This system, which constituted a prerelease of UNIX/32V†, was adopted as the basis for virtual memory extensions.

The virtual memory effort was motivated by several factors in our research environment:

- To provide a very large virtual address space for user processes, in particular, Lisp systems such as MACSYMA, and other systems employed in Image Processing and VLSI design research.
- To provide an easily accessible virtual memory environment suitable for research in the fields of storage hierarchy performance evaluation and automatic program restructuring.
- To try to improve overall system performance by making better use of our very limited memory resource.

The reader should be familiar with the standard UNIX system as described in [RITC 74] and the virtual memory concept in general [DENN 70]. In the next section, we briefly describe the memory-management hardware as it exists in the VAX-11/780 to support virtual memory [DEC 78]. The following sections detail the new kernel operations including new system calls followed by various measurement results.

2. VAX-11/780 Memory-Management Hardware

The VAX-11/780 memory-management hardware supports a two level mapping mechanism to perform the address translation task. The first level page tables reside in system virtual address space and map user page tables. These tables in turn, map the user virtual address space which consists of 512 byte pages. The 32 bit virtual address space of the VAX-11/780 is divided into four equal sized blocks.

---

† UNIX and UNIX/32V are Trademarks of Bell Laboratories

‡ Work supported by the National Science Foundation under grants MCS 7807291, MCS 7824618, MCS 7407644-A03 and by an IBM Graduate Fellowship to the second author.

* VAX and PDP are trademarks of Digital Equipment Corporation.

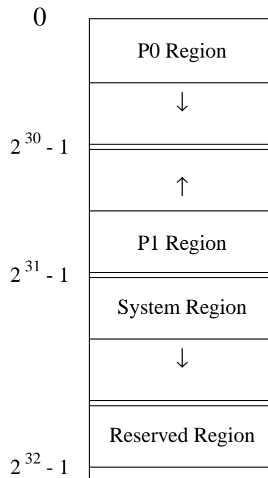

Fig. 2.1. Virtual address space

Two of these blocks, known as the P0 and P1 regions, constitute the two per-process segments. The third block, known as the system region, contains the shared kernel virtual address space while the fourth region is not supported by the current hardware. The P0 segment starts at virtual address 0 and can grow toward higher addresses. The P1 segment on the other hand, starts at the top of user virtual address space and grows toward lower addresses. Both segments are described by two per-process (base, length) register pairs.

A page table entry consists of four bytes of mapping and protection information. Attempting a translation through a page table entry that has the *valid* bit off results in a *Translation Not Valid Fault* (i.e., a page fault). Whereas most architectures that support virtual memory provide a per-page *Reference Bit* that is automatically set by the hardware when the corresponding page is referenced to be examined and/or reset by the page replacement algorithm, the VAX-11/780 has no such mechanism. In order to overcome this deficiency, we distinguish the three states that a page of virtual address space can be in:

[1] Valid – the page is in memory and valid. This corresponds to the *valid, referenced* state in the presence of a reference bit.

[2] Not valid but in memory – the page is in memory but the page table entry is marked not valid so as to cause a page fault upon reference. This is the so called *reclaimable* state of the page. Equivalent to the *valid, not referenced* state.

[3] Not valid and not in memory – the page is in secondary storage. Equivalent to the *not valid* state.

This scheme in effect allows us to detect and record references to pages using software. We discuss the cost and effectiveness of the method in §7.2.

3. Process Structure

In UNIX, the notion of a *process* and a computer execution environment are intimately related [THOM 78]. In fact, a process is the execution of this environment which consists of the process virtual address space state, general register contents, open files, current directory, etc. The state of this pseudo computer is comprised of the contents of four segments. The first three contain the process virtual address space, while the fourth segment describes the system maintained state information.

The process virtual address space consists of a read only program text segment that is shared amongst all processes that are currently executing the same program, as well as private writable data and stack segments. Within the limited segmentation capability of the VAX-11/780, these three segments are mapped such that the program text is in the P0 region beginning at virtual address 0 with the data immediately after it starting at the next page boundary. The stack segment is mapped into the P1 region starting at the highest

virtual address. While the text segment has a static size, the data segment can be grown or shrunk through system calls and the stack segment is grown automatically by the kernel upon the detection of segmentation faults.

The physical structure of these segments in secondary storage (called an *image*) can be organized in various ways. At one extreme is the physically contiguous organization described simply by a (base, length) pair. While appropriate for static segments, such as text, this organization is too rigid for dynamically growing segments, like the data and stack segments. In addition to fragmentation, segment growth beyond the current allocation could imply physical movement of the image. At the other extreme is a fully scattered organization of the image. While minimizing fragmentation, this can result in expensive allocation and mapping functions due to the large number of pages which are present in large images.

The image organization chosen for the dynamic segments represents a compromise between the two extremes. Each image consists of several scattered chunks. An exponentially increasing chunk allocation scheme allows the mapping of very large segments through a small table. Limiting the maximum size of any chunk helps to prevent extreme fragmentation. Thus in the current system, the smallest chunk allocated to a segment is 8K bytes, and chunk sizes increase by powers of two up to a maximum size of 2M bytes.

4. Kernel Functions

We now describe the major new functions performed by the kernel as well as the existing functions whose implementation have been significantly modified. For the purposes of future discussion, we define the following terms:

Free List The doubly linked circular queue of page frames available for allocation. Allocation is always from the head, while insertion occurs both at the head (for pages which can no longer be needed) or the tail (for pages which might still be reclaimed).

Loop Envision the set of physical page frames that are not in the free list as if they were arranged statically around the circumference of a circle. We refer to these set of page frames, ordered by physical address, as the *loop*. Page frames allocated to kernel code and data appear in neither the loop nor the free list.

Hand A pointer to a page frame that is in the loop. The hand is incremented circularly around the loop by the pageout daemon as described below.

4.1. Page Fault Handling

The most visible of the kernel changes is the existence of a *Translation Not Valid* fault handler. Given the virtual address that caused the fault, the system checks to see if the page containing the virtual address is in the *reclaimable* state. This happens when the pageout daemon has swept past a page and made it reclaimable to simulate a reference bit (as described below). If the page is in this state, it can once again be made valid, and the process returns to user mode. Note that if the reclaimed page was in the free list, it is removed and reenters the loop. Since none of the operations involved in reclaiming a page can cause the process to *block*, reclaiming a page does not involve a processor context switch and reschedule.

If the page cannot be reclaimed (i.e., is not no longer in core), then a page frame is allocated and the disk transfer is initiated from the segment image as dictated by the image mapping.

In reality, more cases must be considered. If the faulting page belongs to a shared text segment, the disk transfer is initiated only if the page is not reclaimable and not *intransit*, i.e., the pagein operation has not already been initiated by another process that is sharing the text segment. If intransit, the faulting process sleeps to be waken by the process that started the page transfer when it completes. Here we note that the first level page tables for shared text segments are *not* shared, but rather, each process has its own copy.† Thus, all operations that modify page table entries of shared text segments must insure that all

†Sharing all user level page tables of shared segments would require a 64K byte alignment between the text and data segments. This is not enforced by the current loading scheme, so currently these page tables are not shared at all.

existing copies are updated.

Other types of page faults that require special handling are those where the faulting page is marked as *fill on demand*. There are three types of demand fill pages:

Zero Fill These pages are created due to segment growth and result in a page of zeroes when referenced.

Text Fill These pages result from execution of demand-loaded programs, and cause the corresponding page to be loaded at the point of first reference, from the currently executing object file. Such object files are created by a special directive to the loader and are described further in §5.3.

File Fill These pages are similar to text fill pages, but the pages come from a open file rather than the current text image file. These pages are set up by the **vread** system call. See section §5.2 for more details.

4.2. Page Write Back

During system initialization, just before the *init* process is created, the bootstrapping code creates process 2 which is known as the *pageout daemon*. It is this process that actually implements the page replacement policy as well as writing back modified pages. The process leaves its normal dormant state upon being waken up due to the memory free list size dropping below an upper threshold.

At this point, the daemon examines the page frame being pointed to by the *hand*. If the page frame corresponds to a valid page, it is made reclaimable. Otherwise the page was reclaimable, and it is freed, but remains reclaimable until it is removed from the free list and allocated to another purpose. The hand is then incremented and the above steps are repeated until either the free memory is above the upper threshold or the angular velocity of the hand exceeds a bound.

The rate at which the daemon examines page frames increases linearly between the free memory upper threshold and lower threshold (these are tunable system parameters). In a loaded system the hand will be moved around the loop two to three times a minute.

Upon encountering a reclaimable page that has been modified since it was last paged in, the daemon must arrange for it to be written back before the page frame containing it can be freed. To accomplish this, the daemon queues a descriptor of the I/O operation for the paging device driver. Upon completion of the I/O, the interrupt service routine inserts the descriptor to the *cleaned list* for further processing by the daemon. The daemon periodically empties the cleaned list by freeing the page frames on it that have not been reclaimed in the meantime.

Note that as described above, this pageout process implements a variant of the global CLOCK page replacement algorithm that is known as *scheduled sweep* [CORB 68, EAST 79].

4.3. Process Creation

In UNIX, every process comes into being through a **fork** system call whereby a copy of the *parent* process is created and identified as the *child*. This involves the duplication of the parent's private segments (data and stack) and the system maintained state information (open files, current directory, etc.).

Within a virtual memory environment including the pagein and pageout primitives described above, the implementation of the **fork** system call is conceptually very simple. The parent process copies its virtual address space to the child's one page at a time. Note that this may require faulting in the invalid portions of the parent's address space. Since the VAX-11/780 memory-management mechanism can establish only one mapping at a time, the child's address space is actually created indirectly through the kernel virtual memory.

Although conceptually simple, the above scheme has undesirable system performance consequences. Duplication of the parent's private segments generates a sharp and atypical consumption of memory. Since a significant percentage of all forks serve only to create system contexts to be passed to another process via the **exec** system call, the copying of the parent's private segments is largely unnecessary. The **vfork** system call, described in §5.1, has been introduced to provide an efficient way to create new system contexts within the current design.

4.4. Program Execution

The **exec** system call, whereby a process overlays its address space also has a simple implementation. The process releases its current virtual memory resources and allocates new ones as determined by the program being executed. Then, the program object file is simply read into the process address space which has been initialized as *zero fill on demand* pages so as not to generate irrelevant paging from the process' old image.

This implementation suffers from the same problems as the above fork implementation. Initiation of very large programs is very slow, and results in system wide performance degradation due to the loading of the entire program file in the virtual memory before execution commences. A new loader format which allows demand paging from the program object file has been designed to improve large program start up and to eliminate this non-demand situation (see §5.3).

4.5. Process Swapping

Swapping a process out involves releasing the physical memory currently allocated to it (called the *resident set*) and writing back its modified pages to its image along with the system maintained state information and page tables. Swapping a process in, on the other hand, involves reading in its page tables and state information and resuming it. Note that as no pages from the process address space are brought in, the process will have to fault them back in as required. The alternative of swapping the resident set in and out is not implemented.

4.6. Swap Scheduling

When the amount of available free memory in the system cannot be maintained at a minimal number of free pages by the pageout daemon, then the system invokes the swap scheduler. In order to free memory, the swap scheduler will select a process which is resident and swap it (completely) out. The scheduler prefers first to swap out processes which have been blocked for a significant length of time, and chooses the process which has been in such a state the longest. If there are no such processes, and it is therefore necessary to swap out a process which is or has recently been active, the system chooses from among the remaining processes the one which has been memory resident the longest.

In choosing an active process to swap out, the system checks to guarantee that the process has had a minimal amount of time necessary to demand fault in the number of pages which it had when it was last swapped out (based on maximum expected paging device throughput). This serves to guarantee a minimal amount of progress by a process each time it is swapped in. When a process is forced out while it has many pages, it is given lower priority to return to the set of resident processes than ones which swapped with fewer pages or which are very small.

The swap-in mechanism also recognizes that processes which swapped out with many pages, will need to fault in pages when they are brought in. The system therefore maintains a notion of a global memory *deficit,* which is the expected short term demand for memory from processes recently brought in, based on the number of pages they were using when they swapped out. The deficit is charged against the free memory available when deciding whether to bring a process in.

In general, this swap scheduling mechanism does not perform well under very heavy load. The system performs much better when memory partitioning can be done by the page replacement algorithm rather than the swap algorithm. If heavy swapping is to occur on moving head devices, then better algorithms could be implemented. High speed specialized paging devices, on the other hand, would suggest different algorithms based on migration.

4.7. Raw I/O

In a virtual memory environment, handling input/output operations directly to/from process address space without going through the system buffer cache requires special attention. The pages involved in the I/O must be insured to be valid and locked for the duration of the operation. This is accomplished through the *virtual segment lock/unlock* internal primitives. Locking a virtual segment consists of locking pages that are already valid and faulting/reclaiming invalid pages by simply touching them and refraining from unlocking the page frame (which is allocated in the locked state) after the pagein completion.

5. New System Facilities

5.1. Process Creation

In order to allow efficient creation of new processes, a new system primitive **vfork** has been implemented. After a **vfork,** the parent and its virtual memory run in the child's system context, which may be manipulated as desired for the new image to be created. When a **exec** is executed in the child's context, the virtual memory and parent thread of control are returned to the parent's context and a new thread of control and virtual memory are created for the child. This mechanism allows a new context to be created without any copying.

It should be noted that an implementation of **fork** using a *Copy On Write* mechanism would be completely transparent and nearly as efficient as **vfork.** Such a mechanism would rely on more general sharing primitives and data structures than are present in the current version of the system, so it has not been implemented.

5.2. Virtual Reading/Writing of Files

In order that efficient random access be permitted in a portable way to large data files, a pair of new system calls has been added: **vread** and **vwrite.** These calls resemble the normal UNIX **read** and **write** calls, but are potentially much more efficient for sparse and random access to large data files. **Vread** does not cause all the data which is virtually read to be immediately transferred to the user's address space. Rather, the data can be fetched as required by references, at the system's discretion. At the point of the **vread,** the system merely computes the disk block numbers of the corresponding pages and stores these in the page tables. Faulting in a page from the file system is thus no more expensive than faulting in a page from the swap device. In both cases all the mapping information is immediately available or can be easily computed from in-core information. **Vwrite** works with **vread** to allow efficient updating of large data which is only partially accessed, by rewriting to the file only those pages which have been modified.

Downward compatibility with non-virtual systems is achieved by the fact that **read** and **write** calls have the same semantics as **vread** and **vwrite** calls; only the efficiency is different. Upward extensibility into a more general sharing scheme is also easy to provide, as **vread** can be easily simulated by a mapping of the file into the address space with a copy-on-write mechanism on the pages. Although the current mechanism does not share copies of the same page if it is **vread** twice, the semantics of the system call do not prohibit such an implementation if used with a copy-on-write mechanism. Note that **vwrite** can also be simulated by a map-out-like mechanism.

5.3. New Loader Format

The same mechanism that is used to implement the **vread** system call is used to provide a load format where the pages of the executable image are not pre-loaded, but rather initialized on demand, with the page block numbers only being bound into the page tables at the point of **exec.** The only change from the other UNIX load formats in this new format is the alignment of data in the load file on a page boundary. Large processes using this format can begin execution very quickly, with page faults causing reading from the executed file.

6. Perspective

There are a number of facilities which have not been implemented in the first release of the system as described here.

For example, there are plans to change the system to use 1024 byte disk blocks rather than 512 byte blocks. It has been observed that in many cases the system is limited by the number of disk transactions that can be made per second. Larger disk blocks will help improve disk throughput. On machines with large real memories, using page-pairs in the paging system will also reduce the overhead of the replacement algorithm and increase throughput to the paging device. Since a page contains only 128 words, it does not provide a great deal of locality. It is expected that using 1024 byte pages (in effect) will not degrade the effective memory size significantly.

Another problem associated with the small page size of the VAX architecture is the rate of growth of user page tables.† For very large processes, this not only results in a significant amount of real memory allocated to page tables, but also increases the system overhead in dealing with them. Effectively supporting extremely large virtual address spaces will require handling translation faults at the first level (i.e., page table faults) whereby real memory for page tables is allocated on demand.

The system changes as presented here are the result of approximately one man year of effort. The base system (a prerelease of UNIX/32V that was maintained as the production system during the paging development) and the paged system consist of approximately 14800 and 16800 total source lines, respectively. The break down of these numbers amongst the various types of source is presented in Table 6.1. Also presented is a comparison that excludes comment lines from the source of the two systems.‡ We note that the actual number of lines *modified* to obtain the paged system is considerably more than the simple net difference for each category. In the meantime, for equal configurations, the resident kernel size has increased by about 12K bytes of program and 26K bytes of data resulting in a total size of about 164K bytes (for a 1 megabyte main memory system).

Category	Total Source Lines		Non Comment Lines	
	Base System	Paged System	Base System	Paged System
Assembly Code	1292	1353	1062	1015
C Code	11581	13405	4891	5614
Header Files	1997	2068	1223	1316

Table 6.1. Source Code Volume Comparison

7. Measurement Results

The system has been instrumented to collect data related to various paging system activities as well as workload characteristics in general.

7.1. Process Virtual Size Distribution

Being one of the few quantifiable characteristics of a workload that is also of importance in a virtual memory environment, system-wide distribution of process virtual size was monitored. The results of integrating process data and stack segment size over user CPU time are shown in Fig. 7.1.1. The two sets of measurements taken almost one month apart indicate an increase from 29.6K to 161.7K bytes and 6.8K to 9.8K bytes for the mean data and stack segment sizes, respectively. The October 18 measurement corresponds to early stages of production run of the system and is representative of the pre-virtual memory workload. The significant increase in the average data segment size within this short period is attributed to the rapid growth of Lisp systems including MACSYMA. The insignificant contribution of the stack segment to total process size is a characteristic of our C intensive workload.

7.2. Page Fault Service Time Distribution

As described in §2, a page can be in three states. Reference to a page in memory but invalid causes a *reclaim*, whereas reference to one not in memory results in a *page-in* operation. The service time distributions for these two different types of faults is shown in Fig. 7.2.1. For the reclaim time distribution, the first peak corresponds to reclaims from the *loop*, while the second bump and the long tail are accounted for by the load dependent component of the service time due to reclaiming pages of shared text segments.† Note

†Since a page table entry is 4 bytes, user page tables grow one byte for each 128 bytes of user virtual memory.

‡ For the C source code, the number of occurrences of ";" was used as an estimate of the actual number of source lines that were not comments.

† This operation requires updating the page tables of all processes currently executing the same code, thus varies with load.

Fig. 7.1.1. Process size distribution: (a) data, (b) stack segments

that the mean reclaim time of 208 microseconds per reclaim represents a negligible delay to user programs. Furthermore, the overall system cost of reclaims through which we simulate the missing reference bits of the architecture has been measured to be less than 0.05% of all CPU cycles.‡

The page-in service time distribution is highly load dependent since it includes all of the queueing as well as process rescheduling delays. The configuration with the paging activity on the same arm (an RM-03 equivalent disk) as the temporary and the root file systems results in a 54.9 msec total service time. The significant number of services completed under 20 msec are due to the track buffering capability of the controller being used.

7.3. Comparison with Swap-Based System

In an effort to compare the performance of the system before and after the addition of virtual memory, a script driven workload was run in a stand-alone manner in both systems under identical configurations consisting of a 1 megabyte main memory, an RP-06 servicing the user file system and an RM-03 shared by the root and temporary file systems in addition to the swapping/paging activity. The swap-based system used for this comparison was quite sophisticated, performing scatter loading of processes into memory and partially swapping processes to obtain free memory.

The basic unit of work generated by the script is made up of four concurrent terminal sessions:

lisp A recompilation, using a Lisp compiler, of a portion of the compiler, and a “dumplisp” using the lisp interpreter to create a new binary version of the compiler. Under the paging system, a system load format which caused the interpreter and compiler to be demand loaded (rather than preloaded) was used (cf. §5.3).

ccomp An editor session followed by a recompilation and loading of several C programs which support the line printer.

‡This cost is actually a function of the paging activity. The number reported here has been averaged over a 28 hour period in a 1.25M byte real memory configuration

Fig. 7.2.1. Fault service time distribution: (a) reclaim, (b) page-in

typeset An editor session followed by typesetting of a paper involving mathematical processing, producing output for a Versatec raster plotter.

trivial Repeated execution of a trivial command (printing the date) every few seconds.

Staggered multiple initiations of from one to four of these terminal sessions were used to create increasing levels of load on the system. Figure 7.3.1 gives the average completion time for each category of session under the two systems. For the non-trivial sessions, completion times were very similar under the two systems, with the paging version of the system running (in all but one case) faster, and the swap-based system departing from linear degradation more rapidly. This trend is most noticeable in the response time for the trivial sessions. Systemwide measures collected during the experiments are given in Figure 7.3.2.

Fig. 7.3.1. Average completion times
(a) lisp, (b) typeset, (c) ccomp, (d) trivial

Fig. 7.3.2. Systemwide measurements
(a) total (b) average completion time, (c) system time, (d) total page traffic

These measurements show the same trend for both total and average completion times as for individual sessions, with the paging system slightly faster and degrading more linearly than the swap system within the measured range. System overhead was uniformly greater under the paging system, constituting 26 percent of the CPU utilization as compared to 20 percent under the swap system. User CPU utilization was, however, uniformly greater under the paging system, averaging 48 percent, while the swap-based system averaged only 42 percent.

Finally, the total page traffic generated by the two systems was measured. This accounts for both paging and swapping traffic under the paging system, as well as transfer of all system information (control blocks and page tables) under both systems. Although the paging system resulted in far fewer total pages transferred, the actual number of transactions required to accomplish this was much greater since most data transfers, under the paging system, are due to paging rather than swapping activity, and thus take place in very small (512 byte) quantities. We are currently installing modifications in the system to use larger block sizes in both the file and paging subsystems, and expect improved performance from these changes.

Acknowledgments. The cooperation of Bell Laboratories in providing us with an early version of UNIX/32V is greatly appreciated. We would especially like to thank Dr. Charles Roberts of Bell Laboratories for helping us obtain this release, and acknowledge T. B. London and J. F. Reiser for their continuing advice and support.

We are grateful to Domenico Ferrari, Richard Fateman, Jehan-François Pâris, William Rowan, Keith Sklower and Robert Kridle for their participation in the early stages of the design project, and would like to thank our user community for their patience during the system development period.

References

[CORB 68] F. J. Corbato, ‘‘A Paging Experiment with the Multics System,’’ Project MAC Memo MAC-M-384, July, 1968, Mass. Inst. of Tech., published in *In Honor of P. M. Morse*, ed. Ingard, MIT Press, 1969, pp. 217-228.

[DEC 78] *VAX-11/780 Hardware Handbook*, Digital Equipment Corporation, 1978.

[DENN 70] P. J. Denning, ‘‘Virtual Memory,’’ *Computer Surveys*, vol. 2, no. 3 (Sept. 1970), pp. 937-944.

[EAST 79] M. C. Easton and P. A. Franaszek, ‘‘Use Bit Scanning in Replacement Decisions,’’ *IEEE Trans. Comp.*, vol. 28, no.. 2 (Feb. 1979), pp. 133-141.

[RITC 74] D. M. Ritchie and K. Thompson, ‘‘The UNIX Time-Sharing System,’’ *Commun. Assn. Comp. Mach.*, vol. 17, no. 7 (July 1974), pp. 365-375.

[THOM 78] K. Thompson, ‘‘UNIX Implementation,’’ *Bell System Tech. Journal*, vol. 57, no. 6 (July-Aug. 1978), pp. 1931-1946.