

BEFORE MEMORY WAS VIRTUAL

Peter J. Denning
George Mason University

DRAFT 6/10/96

© Copyright 1996 by Peter J. Denning. You may
make one hard copy for personal use only. For
any additional copies, in any form and for any
purpose, you need my express permission. You
can contact me at pjd@cs.gmu.edu.

BEFORE MEMORY WAS VIRTUAL

Peter J. Denning
George Mason University

DRAFT 6/10/96

Virtual memory, long a standard feature of nearly every operating system and computer chip, is now invading the Internet through the World Wide Web. Once the subject of intense controversy, virtual memory is now so ordinary that few people think much about it. That this has happened is one of the engineering triumphs of the computer age.

Virtual memory is the simulation of a storage space so large that programmers do not need to rewrite programs, and authors do not need to rewrite documents, when the content of a program module, the capacity of a local memory, or the configuration of a network changes. The name, borrowed from optics, recalls the virtual images formed in mirrors and lenses -- images that are not there but behave as if they are. The story of virtual memory, from the Atlas Computer at the University of Manchester in the 1950s to the multicomputers and World Wide Web on the 1990s, is not simply a story of automatic storage allocation, it is a story of machines helping programmers with protection of information, reuse and sharing of objects, and linking of program components.

In what follows, I will give an account of the history of virtual memory, including personal recollections of the events I witnessed and experienced. These recollections will make up a kind of photo album of technical snapshots of virtual memory from birth, through adolescence, to maturity. From these snapshots you will see that the real driver was building a good environment for programming; automatic storage allocation was actually a secondary concern. You will see that virtual memory's designers had deep insights about modularity, sharing, reuse, and objects a long time ago, a fact that may surprise some younger members of the field. Without those insights, virtual memory would never have passed its adolescence.

I was a graduate student at MIT Project MAC during 1964-68, a time of intense debate on the principles to be incorporated into Multics, principles that would make a permanent imprint on operating systems. Automatic storage allocation and management of multiprogramming were high on the agenda. I contributed the working set model for program behavior. The working set model generated a practical theory for measuring a program's dynamic memory demand and building an automatic control system that would schedule processor and memory for optimum throughput and response time. I also contributed a unified picture of virtual memory that helped to settle the controversies surrounding it.

Programming with Memory Hierarchies

From their beginnings in the 1940s, electronic computers had two-level storage systems. The main memory was magnetic cores (today it is RAMs); the secondary memory was magnetic drums (today it is disks). The processor (CPU) could address only the main memory. A major part of a programmer's job was to devise a good way to divide a program into blocks and to schedule their moves between the levels. The blocks were called "segments" or "pages" and the movement operations "overlays" or "swaps". The designers of the first operating systems in the 1950s dreamt of automating all this storage management.

It is easy to see how the programmer's job is significantly impacted by a memory hierarchy. If you write a matrix multiply algorithm straight from the definition in the textbook, you will create a program with three nested loops covering six lines of text. This program becomes *much* more complicated if you cannot fit the three matrices in main memory at the same time: you have to decide which rows or columns of which matrices you can accommodate in the space available, create a strategy for moving them into main memory, and implement that strategy by inserting additional statements into the program. You will come to several conclusions from the exercise: (1) devising an overlay strategy is time consuming, in this example more than programming the guts of the algorithm, (2) the overlay strategy depends on the amount of memory you assume is available, and (3) the size of the program increases by a factor of two or three. Many programmers reached the same conclusions for other programs that will not fit into the main memory.

So it was pretty obvious to the designers of operating systems in the early 1960s that automatic storage allocation could significantly simplify programming. The first operating system design group to accomplish this was the Atlas team at the University of Manchester who, by 1959, had produced the first working prototype of a virtual memory (Fotheringham 1961; Kilburn et al 1962). They called it *one-level storage system*. At the heart of their idea was a radical innovation --- a distinction between "address" and "memory location". It led them to three inventions. (1) They built hardware that automatically translated each address generated by the processor to its current memory location. (2) They devised demand paging, an interrupt mechanism triggered by the address translator that moved a missing page of data into the main memory. (3) They built a replacement algorithm, a procedure to detect and move the least useful pages back to secondary memory.

Virtual memory was adopted widely in commercial operating systems in the 1960s. The IBM 360/67, CDC 7600, Burroughs 6500, RCA Spectra/70, and GE 645 all had it. By the mid 1970s IBM 370, DEC VMS, DEC TENEX, and Unix had had it too. All these systems used it to implement multiprogramming, a stratagem for loading several programs simultaneously into main memory in order to maintain high processing efficiency. Virtual memory solved not only the storage allocation problem but the more critical memory protection problem. Much to everyone's surprise, these systems all exhibited "thrashing", a condition of near-total performance collapse when the multiprogrammed load was too high (Denning 1968). (See Figure 1.) The concern to eliminate thrashing triggered a long line of experiments and models seeking effective load control systems. This was finally accomplished by the late 1970s --- near-optimal throughput will result when the virtual memory guarantees each active process just enough space to hold its working set (Denning 1980).

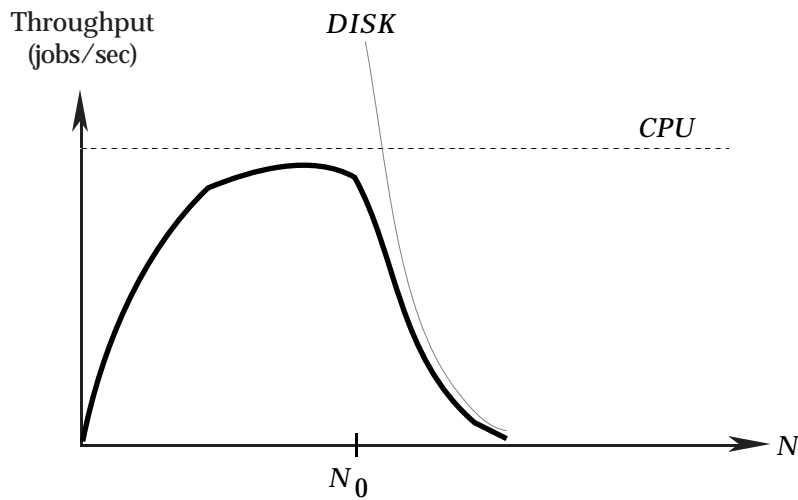


Figure 1. When it was first observed in the 1960s, thrashing was an unexpected, sudden drop in throughput of a multiprogrammed system when the number of programs sharing the main memory N passed a critical threshold N_0 . No one knew how to calculate the threshold, which varied with the changing workload. I explained the phenomenon in 1968 and showed that a working-set memory controller would stabilize the system close to the optimal multiprogramming level. In 1975, many colleagues and I showed from queueing network models that the sudden drop was forced by saturation of the paging disk, whose limit on throughput could be calculated from disk access times and program page-fault-rate functions.

Hardware designers also found virtual memory attractive. In 1965, Maurice Wilkes proposed the “slave memory”, a small high-speed store included in the processor to hold, close at hand, a small number of most recently used blocks of program code and data. Slave memory also used address translation, demand loading, and usage-based replacement. Wilkes said that, by eliminating many data transfers between processor and the main memory, slave memory would allow the system to run within a few percent of the full processor speed at a cost of a few percent of the main memory (Wilkes 1965). The term “cache memory” replaced “slave memory” in 1968 when IBM introduced it on the 360/85 machine. Cache memory is now a standard principle of computer architecture (Hennessey 1990).

Thrashing Solved by The Working Set Model

When I arrived at MIT in 1964, the debate over virtual memory was in full flower. Jack Dennis, who had written extensively and eloquently on the value of segmented address spaces for sharing, modularity, protection, and reusability, had won a place for segmented virtual memory in Multics. Others, such as Richard Kain, liked the elegance of the mechanism, but worried that its cost would be too high and that excessive “page turning” could kill system performance. Jerry Saltzer and Fernando Corbató took a middle ground --- they sought an automatic control system that would regulate the

multiprogrammed memory for optimal throughput. The ideal control would have a single “tuning knob” that would be adjusted once, like the automatic tuning system of an FM receiver.

I took it as a challenge to solve that control problem. The goal was a virtual memory system that could satisfy the programming needs of a Dennis and the performance needs of a Kain. Since fault-rates depend ultimately on the patterns by which programs reference their pages, I began to think extensively about program behavior. My quest was a single model that would account for all the individual cases (loops, nested loops, array referencing, clusters of subroutines, etc). I came to believe that the central property of this model had to be “locality” --- the tendency of programmers to write programs that address subsets of their pages over long time intervals. I purposely put the programmer into this formulation because I was convinced that human thought patterns for problem-solving tended to impose structure and regularity on the dynamics of their programs; in short, locality was a universal property of programs because it is a universal property of how people think. I found many allies for this view in the virtual machine research group at IBM Yorktown labs, especially with Les Belady, who had taken extensive measurements of programs and had found locality in some form in every one.

After many months and many failed models for locality, late one night in the Christmas recess of 1966 I had the Aha! insight that became the working set model. Rather than find a way to *generate* locality, all I needed was a standard way to *observe* locality. I proposed to define the working set as the set of pages referenced in a sampling window extending from the current time backwards into the past. The idea of sampling for used pages was not new; it appeared within usage-bit-based paging algorithms. What was new was that the window was defined in the virtual time of the program --- i.e., CPU time with all interrupts removed -- so that the same program with the same input data would have the same locality measurements no matter what the memory size, multiprogramming level, or scheduler policy. This simple shift of observer made a profound difference. I chose to call the set of pages observed in the window the “working set”, a term that was already being used for the intuitive concept of the smallest set of pages required to be in main memory in order that virtual memory would generate acceptable processing efficiency. The virtual-time-based definition made that precise: the larger the window, the larger the working set, the lower the probability of paging, and the greater the processing efficiency.

In the next few months I discovered that this formulation is extremely powerful than I had first expected. I was able to give formulas for the paging rate and size of a working set memory policy as a function of its single parameter, window size. These basic measures could be computed from the histogram of intervals between successive references to the same page, a function that could be measured efficiently and in real time. I was able to explain in this theory what caused thrashing and how to prevent it. Although the hardware needed to measure working sets dynamically is more expensive than most designers were willing to tolerate, a number of very good approximations based on sampling usage bits were made, and they solved the problem of thrashing. A decade after the original discovery, my student Scott Graham showed experimentally that a single setting of the working set window size would be sufficient to control the entire operating system to be within five or ten percent of its optimum throughput. This provided the final step of the solution to Saltzer and Corbato’s control problem. The basic concepts developed then are widely used today.

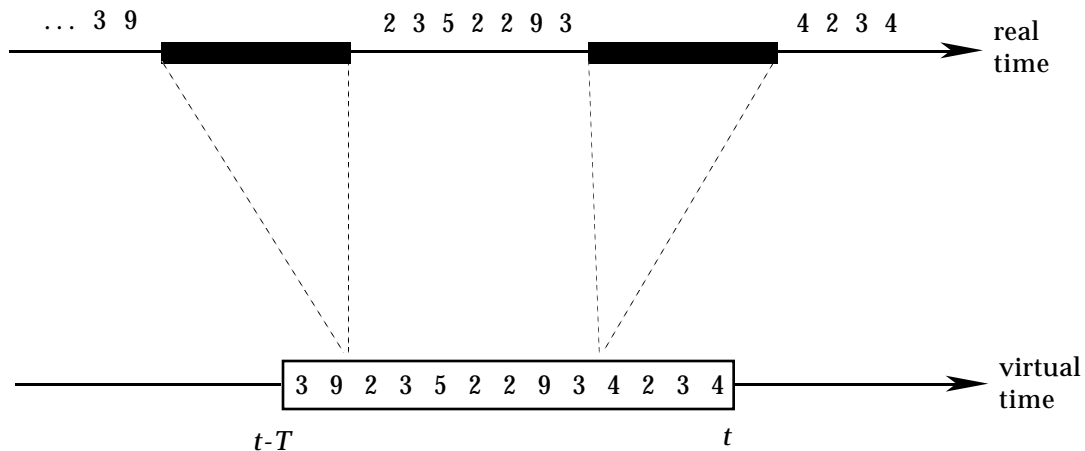


Figure 2. The working set of a program $W(t, T)$ at its virtual time t is the distinct pages referred to by that program in the backward window of size T . The identities of the pages can be determined from usage bits. In this example, the page reference trace inside the window contains the distinct pages $\{2, 3, 4, 5, 9\}$, which is the working set at time t . At two earlier times there were page faults (because pages 2 and 4 may were not present in main memory). The time intervals to service the page faults are shown in black on the real time scale. The processing efficiency is the ratio of black to non-black intervals on the real time scale; it improves as the window size increases.

Memory Management Theory

From 1967 though 1975 I worked on theories of program behavior, memory management, and system performance for computers with virtual memory. The purposes of these models were:

- A program behavior model generates address traces that are statistically close to address traces measured for real programs. Within a model, a small number of parameters can be used to control locality set size, locality holding times, extent of sequential or random correlations, and the like. The model can be used to generate artificial address traces in real time, providing an input for simulators of computer systems. The model can also be used as a basis for system performance models.
- A memory management model computes the number and volume of objects moved per unit time between levels of memory for a given memory management policy.
- A system performance model computes the throughput and response time of a computer system operated under a memory management policy for a workload satisfying given program-behavior assumptions.

I had the privilege of working with many colleagues and students on the models, over 200 in all (Denning 1980). We learned many important things along the way -- for example:

- The principal statistics of a working-set policy can be computed from a histogram of the lengths of intervals between repeated references to pages.
- The average duration of a locality set in a program could be estimated from Little's law applied to the working set: $D = (\text{mean working set size}) / (\text{paging rate})$. D was found experimentally to be relatively insensitive to the window size of the working set policy, giving experimental proof that locality is a real phenomenon.
- Locality in address traces derived from locality of reference in the source code of programs.
- Most programs exhibit phase-transition behavior: long intervals during which all references are restricted to a given locality set. Phases are periods of stability and predictability. Transitions between phases are chaotic.
- A simple law relates system throughput X (jobs completed per second), space-time Y (number of page-seconds of memory consumed by a job), and memory capacity M (number of pages): $M = XY$. Maximizing throughput is the same as minimizing space-time of a job, which is the objective of the memory policy.
- The reciprocal of the fault-rate function of a program gives the mean virtual time between faults; it is called the lifetime function. Allocating a program enough space to accommodate the knee of the function usually minimizes the program's space-time. (See Figure 3.)
- The fault-rate functions of the programs making up the workload become the parameters of a multi-class queueing network model of the computer system on which the work is run. The model allows fast computation of system throughput and response time. It also gives a direct way to compute the upper bound on throughput imposed by saturation of the paging device, giving a simple way to explain thrashing (and avoid it).
- The working set memory management policy is optimal: a single value of working-set window size can be tuned for the whole computer system; it need be dynamically adjusted.

These pursuits took us through a lot of mathematics (probability and queueing theory were the most common) and experimentation (measuring systems and comparing with models). We discovered repeatedly that the stochastic models worked well despite the fact that their key assumptions were usually violated in practice. For example, we discovered that a queueing network model of a system would estimate throughput to within 5% of the value measured for that system, even though the system violated assumptions of statistical steady-state, exponential service times, and random arrivals. In 1975 I entered a collaboration with Jeffrey Buzen, who was trying to explain this phenomenon. He called the approach "operational analysis" to emphasize that models could be formulated from purely operational assumptions. Many limit theorems of queueing theory are operational laws --- relationships among measurable quantities that

hold for every data set (Denning and Buzen 1978). Operational analysis helped us understand why the methods for measuring the statistics of “stack algorithms” and of “working set algorithms” worked so well. (See Appendix for the results of a controversy about operational analysis.)

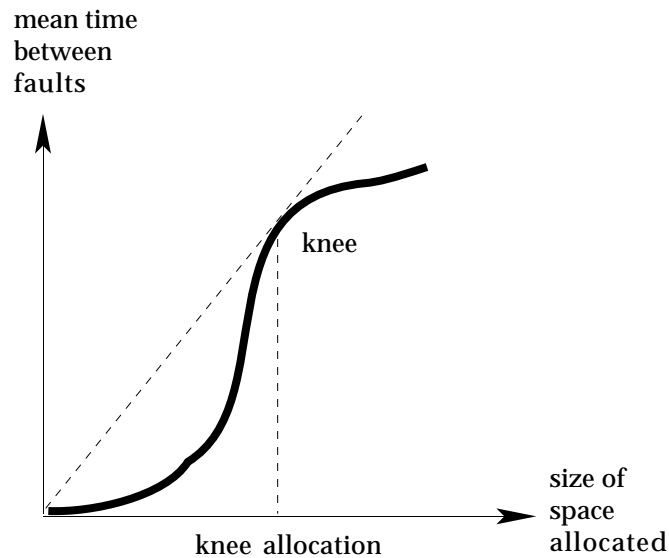


Figure 3. The lifetime curve of a program is the mean time between addressing faults as a function of the mean size of the memory space allocated to it. These function have an overall “S” shape. When the size of the allocated space is close to the knee allocation, the program tends to require the minimum space-time, thus contributing to the system’s capacity to deliver optimum throughput. The working-set policy can be tuned so that it operates every program close to its knee allocation.

The End of the Beginning

The historical thread of concern for program models, memory management, and performance models of computer systems has persisted to the present day as operating system engineers have taken the results into client-server architectures. But to help you understand the rest of the story of the maturation of virtual memory in today’s operating systems, I must return to the beginning and examine the thread of concern for programming.

The literature of 1961 records a spirited debate about automatic storage allocation. By that time, Fortran, Algol, and Lisp had become the first widely-used higher-level programming languages. These languages made storage allocation harder for the programmer because programs were larger, more machine independent, more modular, and their dynamics more dependent on their input data. Dozens of experimental

studies during the 1960s sought either to affirm or deny the hypothesis that operating systems could do a better job at storage allocation than any compiler or programmer.

One by one, the technical problems were solved during the 1960s --- thrashing, optimal memory management, caching, address translation for paging and for segmentation, protected subroutine calls, limited protection domains guarded by hardware. The last remaining part of the debate --- whether the operating system's set of automatic overlay strategies could outperform the best programmers manual overlay strategies --- was laid to rest in 1969 by an extensive study of program locality by David Sayre's IBM research team (Sayre, 1969). Among other things, that team showed that the total number of page moves up and down the memory hierarchy was consistently less with virtual memory than with manual overlays; this meant that virtual-memory controlled multiprogramming systems would be more efficient than manual-controlled systems.

I was able to assemble a coherent picture from all the pieces and I wrote the paper "virtual memory", which went on to enjoy a long period of being regarded as a classic (Denning, 1970). This paper celebrated the successful birth of virtual memory.

Object-Oriented Virtual Memory

If it ended here, this story would already have guaranteed virtual memory a place in history. But the designers of the 1960s were no less inventive than those of the 1950s. Just as the designers of the 1950s sought a solution to the problem of storage allocation, the designers of the 1960s sought solutions to two new kinds of programming problems: (1) Sharable, reusable, and recompilable program modules, and (2) packages of procedures hiding the internal structure of classes of objects. The first of these led to the segmented address space, the second to the architecture that was first called capability-based addressing and later object-oriented programming.

In 1965 the designers of Multics at MIT sought systems to support large programs built from separately compiled, sharable modules linked together on demand. Jack Dennis was the leading spokesman (Dennis 1965, Organick 1972). To Dennis, virtual memory as a pure computational storage system was too restrictive; he held that modular programming would not become a reality as long as programmers had to bind together manually, by a linking loader or makefile program, the component files of an address space. Working with other designers of Multics, Dennis added a second dimension of addressing to the virtual address space, enabling it to span a large collection of linearly-addressed segments. A program could refer to a variable X within a module S by the two-part name (S,X); the symbols S and X were retained by the compiler and converted to the hardware addresses by the virtual memory on first reference (a "linkage fault"). The Multics virtual memory demonstrated very sophisticated forms of sharing, reuse, access control, and protection.

What became of these innovations? The segmented address space did not survive as a recognizable entity, ostensibly because most programmers were content with one, private, linear address space and a handful of open files. The methods of address translation for segmentation were generalized by Dennis and his students into capability machines and, later, object-oriented programming. The dynamic linking mechanism did not die with Multics. It merely went into hibernation, recently reawakening in the guise of the World Wide Web, in which programs and documents contain symbolic pointers to other objects that are linked on demand.

A major turning point occurred in 1965 when Jack Dennis and his student, Earl Van Horn, worked on a paper that they called “programming semantics for multiprogrammed computations.” They devised a generalization of virtual memory that allowed parallel processes in operating systems to operate in their own “spheres of protection”. They replaced the concept of address with a new concept they called “capability”, which was a protected pointer to an arbitrary object. I was Van Horn’s office mate while he worked on this project with Dennis; I remember our blackboard being filled with candidate terms like “pointer”, “handle”, and “generalized address”, and it was some weeks before he and Dennis settled on the word “capability”.

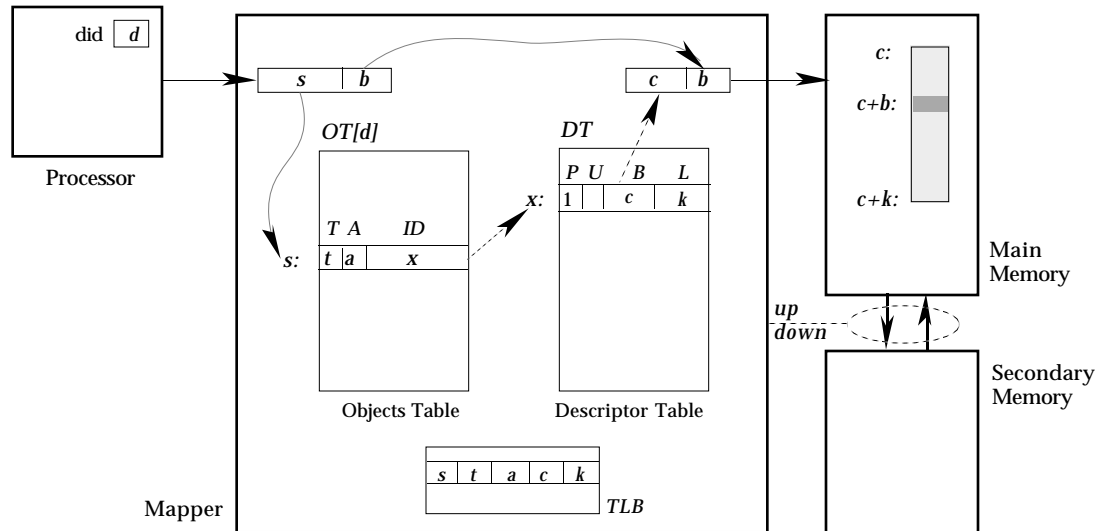


Figure 4. Two-level mapping scheme has become a universal method to translate an address to a reference to a shared object. The domain identifier register (did) associates the processor with an objects table that lists all the objects accessible in that domain. That table converts the local object identifier (s) to the global object identifier (x). The global identifier selects a descriptor, which in this case points to a segment of memory containing the object. A translation lookaside buffer (TLB) holds a number of most-recent paths connecting the local address to the object’s location in memory; the TLB can map repeated references to the object in a small fraction of the time that the two table lookups would take. If an object is not present in main memory, the mapper will issue appropriate up and down commands to the secondary memory. With this scheme, each domain can have a (different) local identifier for the same object, facilitating sharing without prior arrangement; and the physical location of the object is recorded just once, in the descriptor, so that any relocations are effective instantly. In a large distributed system, such as Internet, the local object identifiers are the “URLs” and the global object identifiers “handles”; the main memory is on the local computer and secondary memory is a server somewhere in the network; and the up and down commands correspond to uploads and downloads performed automatically by the network protocol.

Capability Machines

With that extraordinary paper, Dennis and Van Horn initiated a new line of computer architectures, the *capability machines*. They anticipated object-oriented programming: a programmer can build a manager that would create and delete objects of a class and perform operations on them. They introduced the concept called “protected entry point”, a way to call a procedure and force the CPU into a new protection domain associated with that procedure. Dennis and Van Horn were especially concerned that objects be freely reusable and sharable and, at the same time, that their internal structures be accessible only to their authorized managers. Their proposal inspired others to build capability machines during the 1970s, including the Plessey 250, IBM System 38, Cambridge CAP, Intel 432, SWARD, and Hydra. In these systems, capabilities were implemented as long addresses (e.g., 64 bits), which the hardware protected from alteration. (See Fabry 1974, Myers 1982, Wilkes and Needham 1979.) The RISC microprocessor, with its simplified instruction set, rendered capability-managing hardware obsolete by the mid 1980s. But software-managed capabilities, now called *handles*, are indispensable in modern object-oriented programming systems, databases, and distributed operating systems (Chase et al, 1994). The same conceptual structure has also reappeared in a proposal to manage objects and intellectual property in the Internet (Kahn & Wilensky 1995).

You may have wondered why virtual memory, so popular in the operating systems of the 1960s and 1970s, was not present in the personal-computer operating systems of the 1980s. The pundits of the microcomputer revolution proclaimed bravely that personal computers would not succumb to the diseases of the large commercial operating systems; the personal computer would be simple, fast, and cheap. Bill Gates, who once said that no user of a personal computer would ever need more than 640K of main memory, brought out Microsoft DOS in 1982 without most of the common operating system functions, including virtual memory. Over time, however, programmers of personal computers encountered exactly the same programming problems as their predecessors in the 1950s, 1960s, and 1970s. That put pressure on the major PC operating system makers (Apple, Microsoft, and IBM) to add multiprogramming and virtual memory to their operating systems. These makers were able to respond positively because the major chip makers had not lost faith; Intel offered virtual memory and cache in its 80386 chip in 1985; and Motorola did likewise in its 68020 chip. Apple offered multiprogramming in its MultiFinder and virtual memory in its System 6 operating system. Microsoft offered multiprogramming in Windows 3.1 and virtual memory in Windows 95. IBM offered multiprogramming and virtual memory in OS/2.

A similar pattern appeared in the early development of distributed-memory multicomputers beginning in the mid 1980s. These machines allowed for a large number of computers, sharing a high-speed interconnection network, to work concurrently on a single problem. Around 1985, Intel and N-Cube introduced the first hypercube machines consisting of 128 component microcomputers. Shortly thereafter, Thinking Machines produced the first actual supercomputer of this genre, the Connection Machine, with a maximum number of 65,536 component computer chips. These machines soon challenged the traditional supercomputer by offering the same aggregate processing speed at a lower cost (Denning 1990). Their designers initially eschewed virtual memory, believing that address translation and page swapping would seriously detract from the machine's performance. But they quickly encountered new programming problems having to do with synchronizing the processes on different computers and exchanging data among them. Without a common address space, their programmers

had to pass data in messages. Message operations copy the same data at least three times: first from the sender's local memory to a local kernel buffer, then across the network to a kernel buffer in the receiver, and then to the receiver's local memory. The designers of these machines began to realize that virtual memory can reduce communication costs by as much as two-thirds because it copies the data once at the time of reference. Tanenbaum (1995) describes a variety of implementations under the topic of distributed shared memory.

Virtualizing the Web

The World Wide Web (Berners-Lee 1996), extends virtual memory to the world. The Web allows an author to embed, anywhere in a document, a "uniform resource locator" (URL), which is an Internet address of a file. The WWW appeals to many people because it replaces the traditional processor-centered view of computing with a data-centered view that sees computational processes as navigators in a large space of shared objects. To avoid the problem of URLs becoming invalid when an object's owner moves it to a new machine, Kahn and Wilensky have proposed a two-level mapping scheme that first maps a URL to a handle, maps the handle to the machine hosting the object, and then downloads a copy of the object to the local machine (Kahn 1995). This scheme is structurally identical to the mapping of object-oriented virtual memory considered in the 1960s and 1970s (Dennis and van Horn 1966, Fabry 1974). With its Java language, Sun Microsystems has extended WWW links to address programs as well as documents (Gilder 1995); when a Java interpreter encounters the URL of another Java program, it brings a copy of that program to the local machine and executes it on a Java virtual machine. These technologies, now seen as essential for the Internet, vindicate the view of the Multics designers a quarter century ago --- that many large-scale computations will consist of many processes roaming a large space of shared objects.

Conclusion

Virtual memory is one of the great engineering triumphs of the computing age. Virtual memory systems are used to meet one or more of these needs:

1. *Automatic Storage Allocation:* Virtual memory solves the overlay problem that arises when a program exceeds the size of the computational store available to it. It also solves the problems of relocation and partitioning arising with multiprogramming.
2. *Protection:* Each process is given access to a limited set of objects --- its protection domain. The operating system enforces the rights granted in a protection domain by restricting references to the memory regions in which objects are stored and by permitting only the types of reference stated for each object (e.g., read or write). These constraints are easily checked by the hardware in parallel with address formation. These same principles are being used in for efficient implementations of object-oriented programs.
3. *Modular Programs:* Programmers should be able to combine separately compiled, reusable, and sharable components into programs without prior arrangements about anything other than interfaces, and without having to link the components manually into an address space.

4. *Object-Oriented Programs*: Programmers should be able to define managers of classes of objects and be assured that only the manager can access and modify the internal structures of objects (Myers 1982). Objects should be freely sharable and reusable throughout a distributed system (Chase 1994, Tanenbaum 1995). (This is an extension of the modular programming objective.)
5. *Data-Centered Programming*. Computations in the World Wide Web tend to consist of many processes navigating through a space of shared, mobile objects. Objects can be bound to a computation only on demand.
6. *Parallel Computations on Multicomputers*. Scalable algorithms that can be configured at run-time for any number of processors are essential to mastery of highly parallel computations on multicomputers. Virtual memory joins the memories of the component machines into a single address space and reduces communication costs by eliminating much of the copying inherent in message-passing.

From time to time over the past forty years, various people have argued that virtual memory is not really necessary because advancing memory technology would soon permit us to have all the random-access main memory we could possibly want. Such predictions assume implicitly that the primary reason for virtual memory is automatic storage allocation of a memory hierarchy. The historical record reveals, to the contrary, that the driving force behind virtual memory has always been simplifying programs (and programming) by insulating algorithms from the parameters of the memory configuration and by allowing separately constructed objects to be shared, reused, and protected. The predictions that memory capacities would eventually be large enough to hold everything have never come true and there is little reason to believe they ever will. And even if they did, each new generation of users has discovered that its ambitions for sharing objects led it to virtual memory. Virtual memory accommodates essential patterns in the way people use computers. It will still be used when we are all gone.

References

- Berners-Lee, T. 1996. The World Wide Web. *Technology Review* (June).
- Chase, J. S., H. M. Levy, M. J. Feeley, and E. D. Lazowska. 1994. "Sharing and protection in a single-address-space operating system." *ACM TOCS* 12, 4 (November), 271-307.
- Denning, P. J. 1968. "Thrashing: Its causes and prevention." *Proc. AFIPS FJCC* 33, 915-922.
- Denning, P. J. 1970. "Virtual memory." *ACM Computing Surveys* 2, 3 (September), 153-189.
- Denning, P. J. 1980. "Working sets past and present." *IEEE Transactions on Software Engineering* SE-6, 1 (January 1980), 64-84.
- Denning, P. J., and J. Buzen. 1978. "Operation analysis of queueing network models." *ACM Computing Surveys* 10, 3 (September).

- Denning, P. J., and W. F. Tichy. 1990. "Highly parallel computation," *Science* 250 (30 Nov), 1217-1222.
- Denning, P. J. 1996. "Virtual memory". Article in *CRC Handbook of Computer Science and Engineering*. Condensed version in *Computing Surveys* (June).
- Dennis, J. B. 1965. "Segmentation and the design of multiprogrammed computer systems." *JACM* 12, 4 (October), 589-602.
- Dennis, J. B., and E. Van Horn. 1966. "Programming semantics for multiprogrammed computations." *ACM Communications* 9, 3 (March), 143-155.
- Fabry, R. S. 1974. "Capability-based addressing." *ACM Communications* 17, 7 (July), 403-412.
- Fotheringham, J. 1961. "Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store." *ACM Communications* 4, 10 (October), 435-436.
- Gilder, George. 1995. "The coming software shift." *Forbes ASAP* of 8/5/95.
- Hennessey, J. and D. Patterson. 1990. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann.
- Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. 1962. "One-level storage system." *IRE Transactions EC-11*, 2 (April), 223-235.
- Myers, G. J. 1982. *Advances in Computer Architecture*, Wiley. (2nd Ed.)
- Organick, E. I. 1972. *The Multics System: An Examination of Its Structure*, MIT Press.
- Sayre, D. 1969. "Is automatic folding of programs efficient enough to displace manual?" *ACM Communications* 12, 12 (December), 656-660.
- Tanenbaum, A. S. 1995. *Distributed Operating Systems*. Prentice-Hall.
- Wilkes, M. V. 1965. "Slave memories and dynamic storage allocation." *IEEE Trans. EC-14* (April), 270-271.
- Wilkes, M. V., and R. Needham. 1979. *The Cambridge CAP Computer and Its Operating System*. North-Holland.

APPENDIX: OPERATIONAL ANALYSIS

Operational queueing theory was messier than regular queueing theory and was controversial among queueing theorists. A popular criticism was that the operational assumption of homogeneity --- service rates of servers do not depend on system state -- was nothing more than an exponential service-time assumption in disguise. That criticism was neatly dispelled by Ken Sevick and Maria Klawe, who gave an example of an operationally-deterministic system whose throughput and response time were calculated accurately from classical queueing formulas but which in no sense satisfied an exponential service time assumption. Another criticism was that one cannot make predictions of a future system's performance without assuming the present and future systems are manifestations of the same underlying stochastic process. Buzen said that stochastic processes had nothing to do with it; he argued that prediction in practice operates as a process of stating future parameter values and using a validated model to calculate future performance measures. Such small triumphs did little to assuage the critics, who believed that operational analysis denied the existence of stochastic processes.

In 1981, I witnessed a debate between Buzen and one of his critics. I was struck by the symmetry of their arguments. Each started with his domain as the ground and claimed that the other was in effect performing unneeded, error-inducing mappings to get to the same answer. They were both describing the same loop from different angles! This prompted me to write the following little fable.

A Tale of Two Islands

Once upon a time there were two islands. The citizens of Stochasia had organized their society around a revered system of mathematics for random processes. The citizens of Operatia had organized their society around a revered system for experimentation with nondeterminate physical processes. Both societies were closed. Neither would ever have known of the other's existence, had it not been for the events I shall now describe.

At a moment now lost in the mists of antiquity, a great sage of Stochasia posed this problem: Given a matrix of transition probabilities, find the corresponding equilibrium probability distribution of occupying the possible states. He worked out the solution, which he engraved on stones. Ever since, whenever they encounter a problem in life, the Stochasians phrase it in these terms and, using the stones, they find and implement its solution.

At a moment now lost in the mists of antiquity, a great sage of Operatia posed this problem: Having observed a matrix of transition frequencies, calculate the corresponding distribution of proportions of time of occupying the possible states. He worked out the solution, which he engraved on stones. Ever since, whenever they encounter a problem in life, the Operatians phrase it in these terms and, using the stones, they find and implement its solution.

In a recent time there was an anthropologist who specialized in islands. He discovered our two islands from photographs taken by an orbiting satellite. He went to visit Stochasia, where he learned the secrets of their stones. He also visited Operatia, where he learned the secrets of their stones.

Struck by the similarities, the anthropologist asked the elders of each island to evaluate

the approach used by the other island. In due course, each island's elders reached a decision.

The elders of Operatia told the anthropologist: "The Stochasians are hopelessly confused. They have developed a highly indirect approach to solving the problem posed by our great sage. First, they transform the problem into an untestable domain by a process we would call 'abstraction'. Using their stones, they find the abstract answer corresponding to the abstract problem. Finally, they equate the abstract answer with the real world by a process we would call 'interpretation'. They make the audacious claim that their result is useful, even though the two key steps, abstraction and interpretation, can nowise be tested for accuracy. Indeed, these two steps cannot be tested even in principle! Our stones tell us elegantly how to calculate the real result directly from the real data. No extra steps are needed, and nothing untestable is ever used."

The elders of Stochasia told the anthropologist: "The Operatians are hopelessly confused. They have developed a highly indirect approach to solving the problem posed by our great sage. First, they restrict the problem to a single case by a process we would call 'estimation'. Using their stones, they estimate the answer corresponding to their estimate of the problem. Finally, they equate the estimated answer with the real world by a process we would call 'induction'. They make the audacious claim that their result is useful, even though the two key steps, estimation and induction, are nowise error free. Indeed, these two steps cannot be accurate even in principle! Our stones tell us elegantly how to calculate the general answer directly from the parameters. No extra steps are needed, and nothing inaccurate is ever used."

The anthropologist believed both these arguments and was confused. So he went away and searched for new islands.

Some years later, the anthropologist discovered a third island called Determia. Its citizens believe randomness is an illusion. They are certain that all things can be completely explained if all the facts are known. On studying the stones of Stochasia and Operatia, the elders of Determia told the anthropologist: "The Stochasians and Operatians are both hopelessly confused. Neither's approach is valid. All you have to do is look at the real world and you can see for yourself whether or not each state is occupied. There is nothing uncertain about it: each state is or is not occupied at any given time. It is completely determined."

Later, he told this to a Stochasian, who laughed: "That's nonsense. It is well known that deterministic behavior occurs with probability zero. Therefore, it is of no importance. How did you find their island at all?" Still later, he told this to an Operatian, who laughed: "I don't know how to respond. We have not observed such behavior. Therefore it is of no importance. How did you find their island at all?"

The anthropologist believed all these arguments and was profoundly confused. So he went away and searched for new island. I don't know what became of him, but I heard he discovered Noman. (Noman is an island.)