Virtual Memory

Peter J. Denning

March 25, 1986

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report TR-86.8

Research Institute for Advanced Computer Science

Virtual Memory

Peter J. Denning

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report TR-86.8
March 25, 1986

Virtual memory was conceived as a way to automate overlaying of program segments. Modern computers have very large main memories, but need automatic solutions to the relocation and protection problems. Virtual memory serves this need as well and is thus useful in computers of all sizes. This column traces the history of the idea, showing how it has become a widespread, little-noticed feature of computers today.

Work reported herein was supported in part by Cooperative Agreement
NCC 2-387 between the
National Aeronautics and Space Administration (NASA) to the
Universities Space Research Association (USRA).

This is a preprint of the column *The Science of Computing* for
American Scientist 74, No 3, May-June, 1986.

Virtual Memory

Peter J. Denning
Research Institute for Advanced Computer Science

March 25, 1986

In the March-April issue, I described how a computer's storage system is organized as a hierarchy consisting of cache, main memory, and secondary memory (e.g., disk). The cache and main memory form a subsystem that functions like main memory but attains speeds approaching cache. What happens if a program and its data are too large for the main memory?

This is not a frivolous question. Every generation of computer users has been frustrated by insufficient memory. A new line of computers may have sufficient storage for the computations of its predecessor, but new programs will soon exhaust its capacity. In 1960, a long-range planning committee at MIT dared to dream of a computer with 1 million words of main memory. In 1985, the Cray-2 was delivered with 256 million words. Computational physicists dream of computers with 1 billion words. Computer architects have done an outstanding job of enlarging main memories, yet they have never kept up with demand. Only the shortsighted believe they can.

A program that does not fit in main memory must be broken into a collection of blocks, only subsets of which can be in the main memory during phases of the program's execution. The process of creating a schedule for moving blocks between main and secondary memory during program phases is called overlaying. Should the overlay sequence be designed manually by a programmer, or should it be automated?

Designing an overlay sequence for a program can be difficult, even for simple programs and even when the size of the target memory is known in advance. Overlaying is an instance of the general scheduling problem, which is itself difficult. Moreover, detailed technical knowledge of a computer's operating system is often needed for correct use of commands that transfer information between main and secondary memory systems. In 1969 David Sayre of IBM reported a study in which the effort required for manual overlays was significant -- about 40% of the total programming time *(1)*.

The difficulty of designing overlay sequences can be illustrated by a simple program to multiply $n \times n$ matrices, $C = AB$. A short program can be derived from the definition

$$c[i,j] = \sum_{k=1}^{n} a[i,k]\ b[k,j] .$$

This program, however, assumes that A, B, and C are all simultaneously in main memory. Suppose the main memory were large enough to hold only one vector (row or column) from each of A, B, and C? An overlay strategy might

load a row of A , then compute a row of C by loading a column of B for each element; this strategy would add about n (n +2) secondary memory operations to the program's running time. The program would become not only much longer but also more difficult to understand.

Because the human cost of designing overlay sequences is so high, it makes good sense to get the computer to handle the details of overlaying. This fact was recognized by the designers of the earliest computers. By 1949, the design group at the University of Manchester had built an auxiliary memory in which information was stored in fixed-size blocks called pages. By 1959, the Manchester group had built the Atlas computer, in which paging was completely automated; they called their architecture a one-level store because a single address space incorporated both levels of the memory hierarchy *(2)*. Four years later, the B5000 group at Burroughs demonstrated automatic overlaying of small program segments of various sizes. The same technique is used in some modern microcomputers, such as the Intel x808x series or the Z8000. By 1968, the Multics group at MIT had demonstrated automatic overlaying of large segments that were decomposed separately into pages. This collection of architectures came to be called virtual memory because it was based on simulating a large main memory that did not actually exist. I published a survey of the subject in 1970 *(3)*.

The Manchester prototype demonstrated feasibility but did not settle the question whether overlays should be automatic. Indeed, it inspired a heated

debate that lasted nearly fifteen years. Skeptics argued that the extra mechanism was not justified because good programmers could outperform a virtual memory system. They raised many questions that had no immediate answers. In what follows, I'll discuss these questions the ways in which they were ultimately resolved.

Let's begin with a look at the architecture of paging systems, the simplest of virtual memories. The main and secondary memories are divided into equal slots for holding pages of program code or data; the slots of main memory are called page frames. The page is the unit of transfer between the two levels of memory. A page can be placed in any empty frame of main memory without having to relocate anything else. Every page loaded in main memory has a backup copy in the secondary memory. If a page is modified while in main memory, it must be copied back to its secondary-memory slot before being deleted from main memory.

A processor refers to memory by a standard interface consisting of an address register (A), a data register (D), a read control line (R), and a write control line (W). To read from the memory, the processor copies an address into A and sends a signal on R; it waits one memory cycle time for the memory to place the contents of the target location into D; it then copies D to another register. The write operation is similar: the processor puts an address in A and data in D; it signals on W; it waits one memory cycle for the memory to deposit the data into the target location.

A paging system adds more structure inside the memory without changing the standard interface. The memory controller divides the address register into two parts, $A=(H,L)$: the high order bits H are interpreted as a page number and the low order bits L as a line number (byte number) within the page. For example, if A contains 16 bits and L 7 bits, the address space contains 2^{16} (64K) bytes, each page contains 2^7 (128) bytes, and there are 2^9 (512) pages. The memory controller contains a page table T that associates page numbers with frame numbers. Thus $T[H]=F$ records that page H is stored in frame F. The actual hardware address B of the target byte is obtained by replacing the page bits of A with the frame bits: $B=(T[H],L)$. The time to form the address B can be made very small compared to the memory cycle time by holding the page table in a local, very high speed memory. With paging, program components can be scattered randomly through the memory without disturbing the apparent contiguity of the address space seen by the processor.

In practice, the page table is not stored in the memory controller. It is kept in main memory, and the memory controller contains a small, fast cache that holds images of the most recently used page-table entries. This configuration reduces the amount of local memory in the controller, which typically runs at a speed loss under 2%.

Pages can be marked as readable or writable (or both) by storing access bits in page table entries. The memory controller translates H to F only if $T[H]$ allows the requested operation (R or W).

How do we deal with the fact that only some of a program's pages can be physically present in main memory? We augment each entry in the page table with an extra bit, called a present bit, denoted by p. If $p[H]=1$, page H is present in the main memory and if $p[H]=0$, page H is missing. The operating system maintains another table, denoted S, recording the locations of all pages in the secondary memory; thus $S[H]$ is the secondary-memory address of page H. The procedure by which a missing page is moved into main memory can be illustrated by an example. Suppose page 5 is assigned slot 12358 of the secondary memory and is missing from the main memory. Then $S[5]=12358$ and $p[5]=0$. To move page 5 into the main memory, we select some empty frame, say 72, and copy the contents of slot 12358 into it. Finally we set $T[5]=72$ and $p[5]=1$. Now the memory controller will find page 5 when the processor refers to it.

After enough page faults -- that is, requests for missing pages -- the program will fill all available frames with pages. How are frames made available for reuse? A rule called a replacement policy determines a frame containing a least useful page; if that page has been modified since being loaded, it must be copied back out to the secondary memory before the frame can be marked as empty. To assist the replacement policy, the page-table entries must contain bits telling whether a page has been modified or used.

The steps involved in moving a missing page into main memory are too complicated to build into the hardware or microcode of the memory controller.

Instead, they are contained in a special program of the operating system called a page-fault handler. The memory controller treats an attempted reference to a missing page as a fault that must be corrected by this program. The memory controller's page-fault signal is sent to the processor, where the interrupt hardware takes control and causes the processor to begin executing the instructions of the page-fault handler, which is, of course, always loaded in memory. On completion of its duties, the handler returns control to the program that encountered the fault. With the page present, the interrupted instruction can be retried with success.

Let's recapitulate. The paging system consists of five main components: hardware in the memory controller that maps page numbers of processor-generated addresses into frame numbers and verifies permission for read or write access; page-tables, one for each program, that can be stored in the memory controller; a page-fault routine, triggered when the memory controller requests a missing page; a secondary-memory table accessible to the page-fault routine; and a replacement policy. The high ratio of secondary- to main-memory access time, typically on the order of 10^5 memory cycles or higher, makes the performance of this system depend heavily on the replacement policy. Because at least 10^5 memory cycles will elapse before a page fault can be resolved, mistakes by this policy are expensive.

The success of an automatic paging system rests squarely on one question: in the absence of prior knowledge of the algorithm of a program, can practical

replacement policies keep the fault probability at least as low as it would be if a programmer designed the overlay sequence manually? A considerable number of studies in the 1960s and early 1970s tried to devise replacement policies for real programs. The earliest focused on policies for a fixed amount of space in main memory; later studies took up the question of variable allocation under multiprogramming.

The general principle for optimal management of memory is implied by a formula called the space-time law: $\sigma = M / \lambda$, where σ is the average memory space-time per program, M is the total memory available to all programs, and λ is the number of programs completed per second (throughput of the system). The space-time of a program is the area beneath the curve of $n(t)$, the number of pages the program has loaded at time t. For a given amount of memory, the space-time law implies that throughput is at a maximum if and only if the average space-time per program is at a minimum. (The law is easy to derive. Note that $\lambda = c / T$, where c is the number of programs completed by the system in an interval of length T. Then the total space-time over programs completed in T can be written in two equivalent ways: the number of programs multipled by the space-time per program, $c\,\sigma$, and the amount of memory multiplied by the time, MT.)

For memory that is fixed in size, space-time is at a minimum when faults are; the optimal policy is simply to replace the page that will not be used again for the longest time. Practical policies, however, cannot see into the future, and

thus no single replacement policy can minimize space-time for every program. The best hope is a policy that minimizes space-time over ensembles of programs. Which policies meet this requirement is most easily determined by experiment. A wide variety of candidates have been evaluated, including selecting a page at random, selecting the page resident in memory longest, and selecting the page least recently used (LRU). Policies such as the first and second that exploit no usage information are cheap but perform poorly. Policies such as the third that do exploit usage bits in the page table generally do better. The LRU policy has emerged over many studies as the most robust.

The LRU policy does well because of a general law of program behavior called locality. According to this law, a program's execution time (measured in memory references) is divided into natural phases; the address space is divided naturally into segments; and during each phase there is a specific "locality set" of segments that must be loaded in main memory. Programs with long phases have a high degree of locality. For a program with strong locality, a measurement of segments used during a short interval will yield an accurate estimate of the segments likely to be needed in the immediate future. The LRU policy does well on programs with strong locality because the pages of the current locality set will almost certainly be among the most recently used.

David Sayre's landmark study concluded that programs conform to the principle of locality; simple policies like LRU perform well on programs with good locality; and the degree of locality in a program is affected by a

programmer's awareness of the principle *(1)*. Sayre criticized earlier studies, which had tested only programs written for computers without virtual memory. Intended to be fully loaded prior to execution, such programs were unlikely to exhibit strong locality. The correct experiment, he asserted, compares two programming processes. Process A leads from a problem statement to a program containing a handcrafted overlay sequence for a conventional machine. Process B leads from a problem statement to a program organized with the principle of locality in mind.

In an experiment consisting of five problems, Sayre's group ran program A on a machine with the virtual memory turned off, and then ran program B on the same machine with the virtual memory turned on. They found that the virtual memory versions ran slightly longer in real time (about 20%) than the conventional memory versions; produced slightly less page traffic (about 4%); and required significantly less programming effort -- about 15% of the work time was spent organizing algorithms for better locality, as compared to 40% for designing an overlay sequence. The apparent speed advantage of manual overlaying arose because the programmer could overlap page transfers with computation, whereas the virtual memory initiated page transfers on demand only. Because the virtual memory versions generated fewer page transfers, this advantage disappears in multiprogrammed systems. Sayre's study showed virtual memory to be the winner and laid to rest questions about its efficacy.

Parallel to the investigations of the overlay problem were investigations of multiprogramming, a method of simultaneously loading several programs into main memory. Multiprogramming raised two new questions, involving relocation -- how can program components be moved to different places in main memory as they are reloaded? -- and protection -- how can programs be isolated from one another? Virtual memory offers easy answers because each program has its own page table. Relocation is automatic in the address mapper; isolation is automatic because a program can refer only to the pages in its own page table.

Multiprogramming also brought unexpected performance problems. Designers in the 1960s believed that, as the number of programs (N) sharing memory increased, the replacement policy would gradually reduce the space assigned to each program and the system throughput would rise asymptotically to a saturation level. Real systems do not work this way. A critical load N_0 is reached, beyond which throughput drops sharply. This phenomenon is called thrashing. The explanation is that as N increases, the space available to an individual program decreases to the point that finally the program's locality sets can no longer be accommodated. This causes a sharp rise in the program's page-fault rate, transforming the secondary memory into a system bottleneck. (In this respect, multiprogramming resembles life: people commonly complain of having so many tasks that they spend all their time shuffling papers and no time doing real work.) To prevent thrashing, memory policies for multiprogramming must guarantee each program a minimum amount of space and must contain a

load control.

The locality model of program behavior leads naturally to an ideal memory policy, called the working-set policy, that meets both these requirements. I proposed the idea in 1967 and reviewed its consequences in 1980 *(4)*. A program's working set at a given time is defined as the pages used in a fixed, backward-looking window. If the window is the proper size, the working set will be a reliable estimate of the program's locality set most of the time. It turns out that most programs are insensitive to the value of window size; often, one value can be found that will cause most programs in the system to operate within 10% of their space-time minima. Therefore, if the memory policy guarantees each program space for its working set, the system can operate within a few percent of the optimum defined by the space-time law. Moreover, the working-set policy contains an implicit load control: as soon as memory is full of working sets, no more programs can be activated.

Once the community reached a consensus that virtual memory works by taking advantage of locality, there still remained the question, how can we guarantee that programs have strong locality in the first place? Can we do better than Sayre, who merely told programmers about the idea of locality? A considerable effort was made in the early 1970s to restructure programs, clustering segments into pages so that the usage patterns exhibited strong locality. These experiments sought an allocation of segments to pages that does not mask the inherent locality of the program.

The experiments were based on a simple principle. A matrix $[a_{ij}]$ of affinities says that, if segments i and j were assigned to the same page, the program would experience a_{ij} fewer page faults than it would if each segment had its own page. The most successful methods involved measuring affinities by running the program through a simulator of the replacement policy, determining a_{ij} as the number of faults to segment j that occurred when i was already resident in memory under that policy. These methods could reduce the rate of page faults by factors as high as 10 compared to unreorganized compiler output. It should be noted that restructuring methods are irrelevant in systems whose page size is relatively small.

Another group of methods employs directives inserted into program code by a compiler. Prepaging directives, which request loading a page in advance of its first use in a phase, lower space-time by reducing real-time delays for page faults. Prereplacement directives, which request removal of a page after its last use in a phase, lower space-time by reducing the number of pages in memory.

Unfortunately, all these methods are expensive. Like other improvements of code, they are useful only for large programs that are run often enough to justify the cost. Many programs in most systems are small enough to fit in the main memory. Restructuring them would not be effective.

The collective efforts of several hundred researchers and computer architects during the 1960s and early 1970s have led to the widespread use of virtual memory today. Virtual memory is useful in systems of all sizes: it has become a

common feature of the architectures of many large and medium-sized computers and is finding its way into microcomputers. It is an ironic side effect of this development that, while they reduce the need for overlaying, very large main memories increase the need for relocation and protection.

References:

1. Sayre. D. 1969. "Is automatic folding of programs efficient enough to displace manual?" ACM *Communications.* December.

2. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. 1962. "One-level storage system." *IRE Transactions EC-11.* April.

3. Denning, P. J. 1970. "Virtual memory." ACM *Computing Surveys.* September.

4. Denning, P. J. 1980. "Working sets past and present." IEEE *Transactions on Software Engineering.* January.

RIACS

Mail Stop 230-5
NASA Ames Research Center
Moffett Field, CA 94035
(415) 694-6363

The Research Institute for Advanced Computer Science
is operated by
Universities Space Research Association
The American City Building
Suite 311
Columbia, MD 21044
(301) 730-2656