

VIRTUAL MEMORY II

Peter J. Denning
George Mason University

For Computing Surveys

*This is a condensation of an article to appear in
CRC Handbook of Computer Science and Engineering*

12/10/95

Virtual memory is the simulation of a storage space so large that programmers do not need to rewrite programs, or authors 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 designers of the Atlas Computer at the University of Manchester invented virtual memory in the 1950s to eliminate a looming programming problem: planning and scheduling data transfers between main and secondary memory and recompiling programs for each change of size of main memory. Virtual memory is even more useful in the computers of the 1990s, which have more things to hide --- on-chip caches, separate RAM chips, local disk storage, network file servers, large numbers of separately compiled program modules, other computers on the local bus or local network, or the Internet. The story of virtual memory from then to now is a story about machines helping programmers solve problems in storage allocation, protection of information, sharing and reuse of objects, and linking of program components. Virtual memory, common in all computers and operating systems from the smallest microprocessor to the largest supercomputer, is now invading the Internet.

Twenty-five years ago I published an article, "Virtual Memory", in *Computing Surveys* (Denning 1970). At the time, virtual memory was the subject of intense controversies. The conceptual framework I offered helped settle the controversies. 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 designers have three major concerns. (1) Address mapping, the process of translating virtual addresses to memory addresses, should easily accommodate the kinds of objects that programmers are working with. (2) Address translation should be efficient, costing no more than 3% of hardware execution speed. (3) Overall system performance, measured by throughput and response time, should be within 10% of the best possible performance attainable for a given workload.

Mapping

There are many varieties of mechanisms that translate virtual addresses to memory addresses. They depend on whether objects are stored as fixed-size pages or variable-size segments, whether each segment is divided into pages, and whether objects are individually protectable and sharable. All the varieties depend on a two-level table structure of the kind shown in the figure.

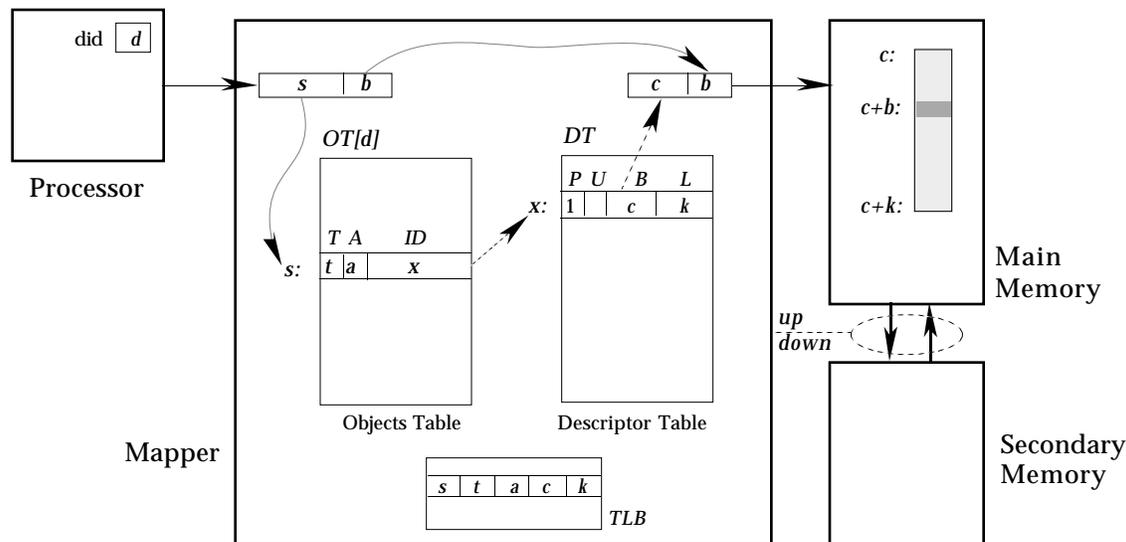


Figure: Object-Oriented Virtual Memory

The mapping from a two-dimensional processor address, (object, byte-within-object), to a one-dimensional memory location address operates in two stages. The first maps an object number *s* to a handle (*t,a,x*) signifying that the object is of type *t*, the accessing process is allowed accesses only of kind *a*, and the object's system-wide unique name is *x*. The second table maps a unique name *x* to a descriptor for the object. The descriptor contains a presence bit with *P=1* meaning the object is in main memory, a usage bit with *U=1* meaning that the object has been recently used, a base address *c*, and length *k* of the main memory region holding the object.

There is one object table for each protection domain; in fact, the domain's object table *defines* the privileges of any process operating within it. A domain identifier register (*did*) in the processor tells the mapper which object table to use. There is only one descriptor for every object; the one, system-wide descriptor table holds them all. A shared object can be listed in several domains, each with its own local object number *s*; all those handles point to the same descriptor. When an object is relocated --- by removing it from main memory or by moving it to a new region of main memory --- only its descriptor is updated to show the change. A translation lookaside buffer (TLB) accelerates mapping by bypassing the tables on repeat accesses to the same object-location path. The mapper's basic operating cycle is:

```

processor places (s,b) in address register
if ((t,a,c,k)=LOOKUP(s) undefined)
  then
    (t,a,x):=OT[i,s]
    (P,c,k):=DT[x]
    if (P=0) then ADDRESS FAULT
    DT[x].U:=1
    LOAD(s,t,a,c,k)
  endif
if (b≥k) then BOUNDS FAULT
if (request not allowed by (t,a)) then PROTECTION FAULT
place c+b in memory address register
  
```

The operation LOOKUP(s) scans all the TLB cells in parallel and returns the contents of the cell whose key matches s . The operation LOAD replaces the least-recently-used cell of TLB with (s,t,a,c,k) . The mapper sets the usage bit U to 1 whenever the entry is accessed so that the replacement algorithm can detect unused objects.

If the TLB already contains the path being attempted, the mapper bypasses the lookups in the object and descriptor tables. In practice, small TLBs (e.g., 64 or 128 cells) give high enough hit ratios that address-translation efficiency goals are easy to meet (Hennessey 1990). The TLB is a powerful and cost-effective accelerator.

Sooner or later the processor will generate an unmapped object address ($P=0$). The mapping unit will detect this and halt, issuing the signal *address fault*. In response, the operating system interrupts the running program and invokes an *address fault handler routine* that (1) locates the needed object in the secondary memory, (2) selects a region of main memory to put that object in, (3) empties that region if need be, (4) copies the needed object into that region, and then (5) restarts the interrupted program, allowing it to complete its reference.

The replacement policy (step 2) frees memory by removing objects. The objective is to minimize “mistakes” --- replacements that are quickly undone when the process recalls the object. This objective is met ideally when the object selected for replacement will not be used again for the longest time among all the loaded objects. A variety of non-lookahead replacement policies have been studied extensively to see how close they come to this ideal in practice. When the memory space allocated to a process is fixed in size, this usually is LRU (least recently used); when space can vary, it is WS (working set) (Denning 1980).

This structure provides the memory partitioning needed for multiprogramming. A process can refer *only* to the objects listed in its object table. The operating system can adjust the size of the main memory region allocated to a process so that the rate of address faults stays within acceptable limits. If too many processes are active at once, the average space available to any one of them will fall below the limit, the average fault rate will overload the queue at the secondary memory device, and the system throughput will drop sharply --- the condition known as thrashing (Denning 1970). System throughput will be near-optimal when the virtual memory guarantees each active process just enough space to hold its working set (Denning 1980).

With virtual memory, the operating system can restrict every process to a domain of least privilege. Only the objects listed in a domain’s object table can be accessed by a process in that domain, and only then in accord with the access codes stored in the object’s handle. In effect, the operating system walls each process off, giving it no chance to read or write the private objects of any other process. This has important benefits for system reliability. Should a process run amok, it can damage only its own objects: a program crash does not imply a system crash. This benefit is so important that many systems use virtual memory even if they allocate enough main memory to hold a process’s entire address space.

The WWW: Virtualizing the Internet

The World Wide Web extends virtual memory to the world. The Web allows an author to embed, anywhere in a document, a “universal resource locator” (URL), which is an Internet address of a file. By clicking the mouse on a URL string, the user triggers the operating system to map the URL to the file and then bring a copy of that file from the

remote server to the local workstation for viewing. The WWW appeals to many people because it replaces the traditional processor-centered view of computing with a data-centered view that view sees computational processes as navigators in a large space of shared objects.

A URL is invalidated when the object's owner moves or renames the object. To overcome this problem, Kahn and Wilensky have proposed a scheme that refers to mobile objects by location-independent "handles" and, with special servers, tracks the correspondence between handles and object locations (Kahn 1995). Their method is equivalent to that described earlier in the figure: first it maps a URL to a handle and then it maps the handle to the Internet location of the object.

The WWW is being extended to programs as well as documents. Sun Microsystems has taken the lead with its Java language. The URL of a Java program can be embedded in another program; exercising the link brings the Java program to a local interpreter, which executes it. The Java interpreter is encapsulated so that imported programs cannot access local objects other than those given it as parameters.

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:* Solving the overlay problem that arises when a program exceeds the size of the computational store available to it. Also includes 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 the main computation. 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.* Scaleable 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 some of the copying inherent in message-passing.

Virtual memory, once the subject of intense controversy, it 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 accommodates essential patterns in the way people use computers.

References

- 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. 1970. "Virtual memory." *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.
- Hennessey, J. and D. Patterson. 1990. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann.
- Kahn, R., and R. Wilensky. 1995. "A framework for distributed digital object services." Technical Note 95-01, Corporation for National Research Initiatives. <<http://www.cnri.reston.va.us>>.
- Myers, G. J. 1982. *Advances in Computer Architecture*, Wiley. (2nd Ed.)
- Tanenbaum, A. S. 1995. *Distributed Operating Systems*. Prentice-Hall.
- Wilkes, M. V., and R. Needham. 1979. *The Cambridge CAP Computer and Its Operating System*. North-Holland.