VIRTUAL MEMORY

Peter J. Denning, Naval Postgraduate School, Monterey, California

January 2008
Rev 6/5/08

Abstract: Virtual memory is the simulation of a storage space so large that users do not need to recompile their works when storage configurations change. Every byte of the virtual memory is addressed in the same way, regardless of the placement of address pace components in the memory hierarchy.

Keywords: operating systems, address mapping, address space, address tr5anslation, protection, capability machines, object-oriented systems, replacement algorithm, working set, locality

Virtual memory is the simulation of a storage space so large that users do not need to recompile their works when 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 paged virtual memory in the 1950s to eliminate two looming problems: planning and scheduling data transfers between main and secondary memory and recompiling programs for each change of size of main memory.

For the first decade after its invention, virtual memory was the subject of intense controversies (1). It significantly improved programming productivity and ease of use, but its performance was unpredictable and it thrashed under multiprogramming. These problems were solved by the 1980s (2). Virtual memory is now so ordinary that few people think much about it. It is one of the engineering triumphs of the computer age.

One of the early lines of virtual memory accommodated objects of various sizes, stored in distinct storage segments. This line produced the first proposal for an object oriented operating system (3), which led to a class of machines called capability machines (4,5) and even to a computer architecture for general object programming (6). The development of RISC produced such speeds that the special hardware in capability machines and their successors was not needed. However, all the methods used in these systems for mapping objects to their locations, protecting objects, and partitioning memory are at the heart of modern object-oriented runtime systems. We will therefore discuss virtual memory from an object point of view.

Virtual memory is ubiquitous in networked systems, which have many things to hide --- on-chip caches, separate RAM chips, local disk storage, network file servers, many separately compiled program modules, multiple computers on the network, and the Internet.

Mapping

The heart of virtual memory is a mapping between an address space and the real memory. The address space is a set of addresses sufficient to name all components of a program independent of their locations in the memory hierarchy. Virtual addresses do not change as objects are moved dynamically to various real addresses within the memory system. Programmers and users see only the virtual address space; the details of the real memory system are hidden.

Most early virtual memories were based on paging. A page is a fixed-size block or program code or data. Main and secondary memory are divided in slots of the same fixed size. Pages can then be moved from any memory slot to any other. Paging yields the simplest form of mapping but wastes storage in the last page assigned to the object.

Some early virtual memories were based on segmentation. A segment is a set of contiguous storage locations of any length. Segments could be sized as exact matches to objects such as procedures and arrays, but they are more difficult than pages to place. Segmentation has become common with object-oriented programming.

The method of mapping virtual addresses to real addresses is basically the same for both fixed and variable sized objects (paging and segmentation). The associations between virtual and real addresses are held in mapping tables; the hardware looks up the current real address for any virtual address generated by the processor. A schematic diagram is shown in Fig. 1.

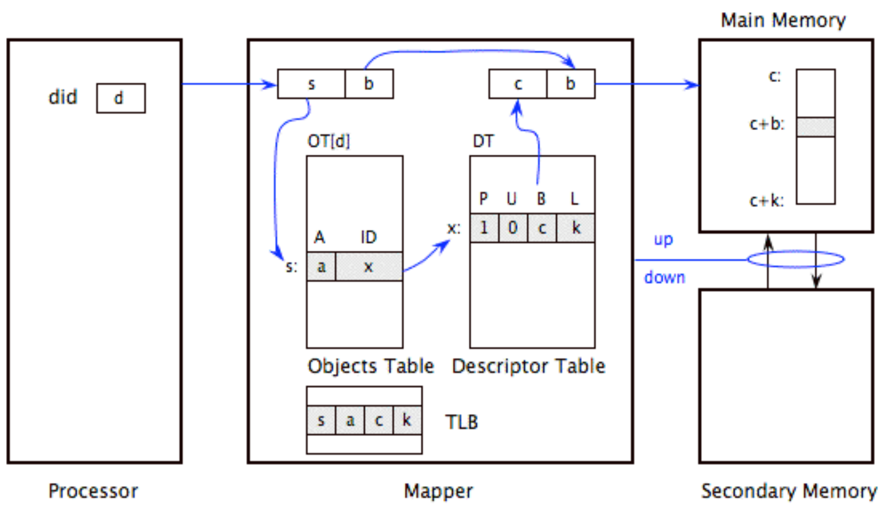

Figure 1. Object-oriented virtual memory

Figure 1 shows the processor on the left and the memory system on the right. The processor generates virtual addresses from the address space of the process it is running. Virtual addresses are of the form (s,b), meaning byte b within segment s. Objects are stored as contiguous segments in the main and secondary memories. Figure 1 shows a segment of k bytes stored at real address c.

Between the processor and the memory is a device called a mapper. Its job is to translate virtual addresses into their current real addresses. For objects already loaded into main memory, translation consists of table lookups that yield the real address. For objects not loaded, the mapper first issues an "up" command to move it from its secondary memory file to an unused segment of main memory; and then it performs the translation. If main memory is full, the mapper will also issue "down" commands as needed to copy loaded objects back to their files and free up their space.

The mapper employs two types of tables, the Descriptor Table (DT) and the Object Tables (OT). Consider first the Descriptor Table. It has one entry for each object. Each object has a unique, system-wide name x. The entry for object x in this table consists of four parts:

- *Presence Bit*: P=1 means the object is loaded in main memory; P=0 means not.
- *Usage Bit*: U=1 means the object has been modified since being loaded; U=0 means not. Modified objects need to be copied back to their secondary files before their space can be released.
- *Base*: The base address of the segment in main memory.
- *Length*: The length of the segment in main memory.

The descriptor table is the only table in the system containing information about the physical locations of objects. When an object is moved, only the descriptor table must be updated.

The second table used by the mapper is an Object Table. There are actually multiple object tables, one for each process. A process's address space is called its "domain", and each address space has a unique domain identifier d. A register in the processor displays the domain of the current process. When the processor switches to a different process, its domain identifier register (did) is automatically changed. Thus, the processor always directs its addresses only to the objects of the currently executing process.

An object table has an entry for each segment s of its address space. That entry contains a handle of two parts:

- *Access Code*: A designates the allowable types of access, for example, read or write.
- *Identifier*: ID contains the unique system identifier x for the object.

The translation of a virtual address (s,b) to the real address containing the byte is straightforward:

1. From OT[d], get the handle for s and extract its identifier x.
2. From DT, get the base c from the descriptor for x.
3. Pass the real address c+b to the memory. (If b≥k, stop with an error.)

The object and descriptor tables are actually stored in a reserved area of main memory belonging to the mapper. Therefore, the table lookups require extra memory accesses. Those extra accesses could slow the system down a fraction of the speed without the mapper.

To bypass the table lookups whenever possible, the mapper contains a device called the Translation Lookaside Buffer (TLB), or address cache. It is a small very high-speed associative memory that holds the most recently used mapping paths. If segment s has been addressed recently, there will be an entry (s,A,B,L) in the TLB. It is built from the A-field of OT[d] plus the B- and L-fields of DT.

The two table lookups are bypassed and are replaced with one ultra-fast TLB lookup.

The mapper's basic cycle is as follows:

```
processor places (s,b) in address register
if ((a,c,k)=LOOKUP_TLB(s) undefined)
   then
      (a,x):=OT[d,s]
      (p,c,k):=DT[x]
      if p=0 then MISSING FAULT
      DT[x].U:=1
      LOAD_TLB(s,a,c,k)
endif
if (b≥k) then BOUNDS FAULT
if (request not allowed by a) then PROTECTION FAULT
place c+b in memory address register
```

The operation `LOOKUP_TLB(s)` scans all the TLB cells in parallel and returns the contents of the cell whose key matches `s`. The operation `LOAD_TLB` replaces the least-recently-used cell of TLB with `(s,a,c,k)`. The mapper sets the usage bit `U` to 1 whenever the entry is accessed.

If the TLB already contains the path requested, 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 (7). The TLB is a powerful and cost-effective accelerator.

Faults

A fault is a condition that prevents additional processing. The mapper can generate three faults: missing object, out of bounds, and protection violation. Those three fault signals trigger the operating system to execute corresponding fault-handler routines that take corrective action.

The bounds fault and protection fault are fatal. References outside a segment are prohibited because they might read or write memory allocated to other objects. Unauthorized references of the wrong kind are also prohibited -- for example attempting to write into a read-only object. The fault handlers for these two faults generally abort the running process.

The missing object fault occurs when the mapper encounters a not-present bit (P=0). The operating system interrupts with a missing object fault 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 by copying its contents to the secondary memory,
4. Copies the needed object into that region,
5. Updates the descriptor table, and then
6. Restarts the interrupted program, allowing it to complete its reference.

Performance

The replacement policy is invoked by the missing object handler at step 2. The performance of virtual memory depends critically on the success of the replacement policy. Each missing object fault carries a huge cost: Accessing the object in main memory might take 10 nanoseconds while retrieving it from secondary memory might take 10 milliseconds -- a speed differential of 100,000. It does not take very many missing object faults to seriously slow a process running in a virtual memory.

The ultimate objective of the replacement policy is to minimize the number of missing object faults. To do this, it seeks to minimize "mistakes" -- replacements that are quickly undone when the process refers to those objects again. 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. Unfortunately, the ideal cannot be realized because we have no way to look ahead into the future. 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) (2).

The operating system can adjust the size of the main memory region allocated to a process so that the rate of missing object faults stays within acceptable limits. System throughput will be near-optimal when the virtual memory guarantees each active process just enough space to hold its working set (2).

Protection

This structure provides the memory partitioning needed for multiprogramming. A process can refer *only* to the objects listed in its object table. It is impossible for a process to accidentally (or intentionally) read or write objects in another address space.

This structure also allows the operating system to 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 Internet. 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 URLs thus act as virtual addresses, and the combination of a server's IP address and a local file path name is the real address.

A URL is invalidated when the object's owner moves or renames the object. This can present operational problems to people who link to that object and depend on its presence. To overcome this problem, Kahn and Wilensky 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 Fig. 1: First it maps a URL to a handle, and then it maps the handle to the Internet location of the object.

Conclusion

Virtual memory is one of the great engineering triumphs of the computing age. Virtual memory systems meet one or more of the following needs:

Automatic Storage Allocation: Solving the overlay problem that originates when a program exceeds the size of the main memory available to it. Also solves the relocation and partitioning problems that develop with multiprogramming.

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. The same principles are used for efficient implementations of object-oriented programs.

Modular Programs: Programmers prepare codes as separately compiled, reusable, and sharable components into programs; their internal structure is hidden behind a public interface. Linker programs combine separate modules into a single address space.

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 (6). Objects should be freely sharable and reusable throughout a distributed system (9,10). Virtual memory mappings are designed for these objectives.

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.

Parallel computations on multicomputers: Scaleable algorithms that can be configured at runtime for any number of processors are essential to mastery of highly parallel computations on clusters of computers. Virtual memory can join the memories of the component computers into a single address space and can reduce communication costs by eliminating some of the copying inherent in message-passing. This is known as distributed virtual memory (10).

BIBLIOGRAPHY

1. P. J. Denning, Virtual memory, *Comput. Surv.* **2**(3): 153-189, 1970.

2. P. J. Denning, Working sets past and present, *IEEE Trans. Softw. Eng.* **SE-6**(1): 64-84, 1980.

3. J. B. Dennis and E. C. Van Horn, Programming semantics for multiprogrammed computations, ACM *Commun.* **9**(3): (March): 143-155, 1966.

4. R. Fabry, Capability based addressing, ACM *Commun.* **17**(7): 403-412, 1974.

5. M. V. Wilkes and R. Needham, *The Cambridge CAP Computer and Its Operating System*. Amsterdam, The Netherlands: North-Holland, 1979.

6. G.J. Myers, *Advances in Computer Architecture,* 2nd ed. New York: Wiley, 1982.

7. J. Hennessey and D. Patterson, *Computer Architecture: A Quantitative Approach*. New York: Morgan-Kaufmann, 1990.

8. R. Kahn and R. Wilensky, A framework for distributed digital object services. Technical Note 95-01, Corporation for National Research Initiatives. Available: http://www.cnri.reston.va.us, 1995.

9. J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, Sharing and protection in a single-address-space operating system, ACM *TOCS* **12**(4): 271-307, 1994.

10. A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms.* Englewood Cliffs, NJ: Prentice-Hall, 2006.

FURTHER READING

P. J. Denning, Virtual memory, *Comput. Surv.* **28(**4): 213-216, 1996.