VIRTUAL MEMORY

Prof. Bruce Jacob
Dept. of Electrical & Computer Engineering
University of Maryland at College Park
http://www.ece.umd.edu/~blj/

Virtual memory is a model—one of many possible models—for managing the resource of physical memory, also called main memory. Such management is necessary because a microprocessor, the heart of a computer, has direct access only to main memory, though all programs and data are stored on permanent media such as hard disks. Reading or writing main memory is as simple as executing a single computer instruction; in contrast, any access to hard disks, DVDs, CD-ROMs, or floppy disks is indirect and requires relatively complex communication protocols involving dozens to thousands of computer instructions. Therefore, accessing a file or running a program requires that the data on disk first be moved into main memory. Virtual memory is one method for handling this management of data, and to illustrate what it is and how it differs from other possibilities, we will compare several alternatives. Details on its mechanisms can be found in the suggested readings listed at the end of the section, in particular two recent surveys by Jacob & Mudge (1998a, 1998b).

Memory-Management Alternatives

The simplest model of program creation and execution is perhaps one in which a programmer determines what physical memory is available, reserves it so that no one else uses it, and writes a program to use the memory locations reserved. This allows the program to execute without any run-time support required from the operating system and is therefore very low-overhead. However, this requires a rewrite of the program every time it runs—a tedious task, but one that could be left to the operating system. At process start-up, the operating system could modify every pointer reference in the application (including loads, stores, and absolute jumps such as function calls) to reflect the physical address at which the program is loaded. The program as it resides in main memory references itself directly and so contains implicit knowledge about the structure and organization of the physical memory. This model is depicted in Figure 2; a program sees itself exactly as it resides in main memory.

A second model is to write the program once using pointer addresses that are not absolute but are instead relative offsets from the beginning of the program. If the location of the program in physical

memory is a variable stored in a known hardware register, then at run-time one can load the program wherever it fits in physical memory and place this location information in the known register so that all memory references first incorporate this base address and are then redirected to the correct physical location. This model is depicted in Figure 3; a process sees itself as a contiguous region or set of contiguous regions, each with its physical location stored in a known hardware register. The advantage is that, as opposed to the previous scheme, knowledge of the memory-system organization is not exposed to the program. The disadvantage is that the program must be divided into a relatively small number of contiguous segments, and each segment must fit entirely in main memory if it is to be used.

A third model is to write the program as if it is loaded at physical memory location zero, load the program wherever it fits (not necessarily location zero), and use some as yet undefined mechanism to translate the program's addresses to the equivalent physical addresses while it is running. If the translation granularity is relatively small (i.e. the program is broken down into smaller pieces that are translated independently of each other), the program can even be fragmented in main memory—bits and pieces of the program can lie scattered throughout main memory, and the program need not be entirely resident to execute. This model is depicted in Figure 4. The advantage of this scheme is that one never needs to rewrite the program. The disadvantage is the potential overhead of the translation mechanism.

The three models can be called physical addressing, base+offset addressing, and virtual addressing. Physical addressing can be implemented on any hardware architecture, base+offset addressing can be implemented on any architecture that has the appropriate addressing mode or address translation hardware, and virtual addressing is typically implemented on microprocessors with memory-management units (MMUs). The following paragraphs discuss the relative merits of the three models.

Physical Addressing. In physical addressing, program execution behaves differently (in that different addresses are used) every time the program is executed on a machine with a different memory organization, and it is likely to behave differently every time it is executed on the same machine with the same organization, since the program is likely to be loaded at a different location every time. Physical addressing systems outnumber virtual addressing systems: an example is the operating system for the original Macintosh, which did not have the benefit of a memory-management unit (Apple Computer, Inc. 1992). Though newer Macintosh systems have an optional virtual memory implementation, many applications require that the option be disabled during their execution for performance reasons. The newest version of the Macintosh operating system, Mac OS X (Apple Computer, Inc. 2000), is based on a Unix core and has true virtual memory at its heart.

The advantages of the physically-addressed scheme are its simplicity and performance. The disadvantages include slow program start-up and decreased flexibility. At start-up, the program must be edited to reflect its location in main memory. While this is easily amortized over the runtime of a long-running program, it is not clear whether the speed advantages outweigh this initial cost for short-running programs. Decreased flexibility also can lead to performance loss; since the program cannot be fragmented or partially loaded, the entire program file must be read into main memory to execute. This can create problems for systems with too little memory to hold all the running programs.

Base+Offset Addressing. In base+offset addressing, like physical addressing, program execution behaves differently every time the program is executed. However, unlike physical addressing, base+offset addressing does not require a re-write of the program every time it is executed. Base+offset systems far outweigh all other systems combined: an example is the DOS/Windows system running on the Intel x86 (Duncan et al. 1994). The Intel processor architecture has a combined memory management unit that places a base+offset design on top of a virtual addressing design. The architecture provides several registers that hold “segment” offsets, so a program can be composed of several regions, each of which must be complete and contiguous, but that need not touch each other.

The advantages of this scheme are that the code needs no editing at process start-up and that the performance is equal to that of the physical addressing model. The disadvantages of the scheme are similar to physical addressing: a region must not be fragmented in main memory, which can be problematic when a system has many running programs scattered about main memory.

Virtual Addressing. In virtual addressing, program execution behaves identically every time the program is executed, even if the machine’s organization changes, and even if the program is run on different machines with wildly different memory organizations. Virtual addressing systems include nearly all academic systems, most Unix-based systems such as Mac OS X (Apple Computer, Inc. 2000), and many Unix-influenced systems such as Windows NT (Custer 1993) and Windows 2000.

The advantages of virtual memory are that a program needs no re-write on start-up, one can run programs on systems with very little memory, and one can easily juggle many programs in physical memory because fragmentation of a program’s code and data regions is allowed. In contrast, systems that require program regions to remain contiguous in physical memory might become unable to execute a program because no single unused space in main memory is large enough to

hold the program, even if many scattered unused areas together would be large enough. The disadvantage of the virtual addressing scheme is the increased amount of space required to hold the translation information and the performance overhead of translating addresses. These overheads have traditionally been no more than a few percent.

Now that the cost of physical memory (i.e. DRAM) has decreased significantly, the schemes that waste memory for better performance—physical and base+offset addressing—have become better choices. Because memory is cheap, perhaps the best design is now one that simply loads every program entirely into memory and assumes that any memory shortage will be fixed by the addition of more DRAM. However, the general consensus is that virtual addressing is more flexible than the other schemes, and we have come to accept its overhead as reasonable. Moreover, it (arguably) provides a more intuitive and bug-free paradigm for program design and development than the other schemes.

How Are Virtual Addresses Translated?

In the virtual addressing model, programs execute in imaginary address spaces that are mapped onto physical memory by the operating system and hardware; executing programs generate instruction fetches and loads and stores using imaginary or "virtual" addresses for their instructions and data. The ultimate home for the program's address space is backing store, usually a disk drive; this is where the program's instructions and data originate and where all of its permanent changes go. Every hardware memory structure between the CPU and the backing store is a cache—temporary storage—for the instructions and data in the program's address space. This includes main memory: main memory is nothing more than a cache for a program's virtual address space. Everything in the address space initially comes from the program file stored on disk or is created on demand and defined to be zero. Figure 5 illustrates.

In Figure 5(a) the program view is shown; a program simply makes data loads and stores and implicit instruction fetches to its virtual address space. The address space, as far as the program is concerned, is contiguous and held completely in main memory, and any unused holes between objects in the space are simply wasted space. In Figure 5(b) we see a more realistic picture; there is no linear storage structure that contains a program's address space, especially since every address space is at least several gigabytes when one includes the unused holes. The address space is actually a collection of fixed-sized "pages" that are stored piecemeal on disk and conjured up out of thin air; the instructions and initialized data can be found in the program file, and when the running program needs extra workspace the operating system can dynamically allocate new pages in main memory.

The enabling mechanism is the page table, a database managed by the operating system that indicates, for every page in a program's address space, whether the page is found on disk, or needs to be created from scratch, or can be found in physical memory at some location. Every virtual address generated by the program is translated according to the page table before the request is sent to the memory system. To speed access to the page table, parts of it are held temporarily in hardware (this is one of the functions of a memory-management unit). To find out more about page table organizations and hardware support for virtual memory, see the two articles listed below by Jacob and Mudge (1998a, 1998b).

In conclusion, virtual memory is but one of many models of program creation and execution, one of many techniques to manage one's physical memory resources. Other models include base+offset addressing and physical addressing, each of which offers performance advantages over virtual addressing at a cost in flexibility. The widespread use of virtual memory in contemporary operating systems is testimony to the fact that flexibility is regarded as a system characteristic with much value—value that outweighs any small amount of performance loss.

Byline: Dr. Bruce Jacob, University of Maryland at College Park, http://www.ece.umd.edu/~blj/

Final word count: 1980

Acknowledgments

Bruce Jacob is supported in part by the National Science Foundation under NSF CAREER Award CCR-9983618 and NSF grant EIA-0000439, by the Department of Defense under the MURI award AFOSR-F496200110374, and by Compaq and IBM.

Bibliography

Apple Computer, Inc. Technical Introduction to the Macintosh Family, 2nd ed. Reading, MA: Addison-Wesley Publishing Company, 1992.

Apple Computer, Inc. Inside Mac OS X: System Overview. Cupertino, CA: Apple Computer, Inc., 2000.

Custer, Helen. Inside Windows NT. Redmond, WA: Microsoft Press, 1993.

Duncan, Ray, et al. Extending DOS – A Programmer's Guide to Protected-Mode DOS, 2nd ed. Reading, MA: Addison-Wesley Publishing Company, 1994.

Jacob, Bruce, and Trevor Mudge. "Virtual memory: Issues of implementation." IEEE Computer 31, no. 6 (1998):33-43. June 1998a. <http://www.ece.umd.edu/~blj/papers/computer31-6.pdf>

——. "Virtual memory in contemporary microprocessors." IEEE Micro 18, no. 4 (1998):60-75. July/August 1998b. <http://www.ece.umd.edu/~blj/papers/micro18-4.pdf>

Figures

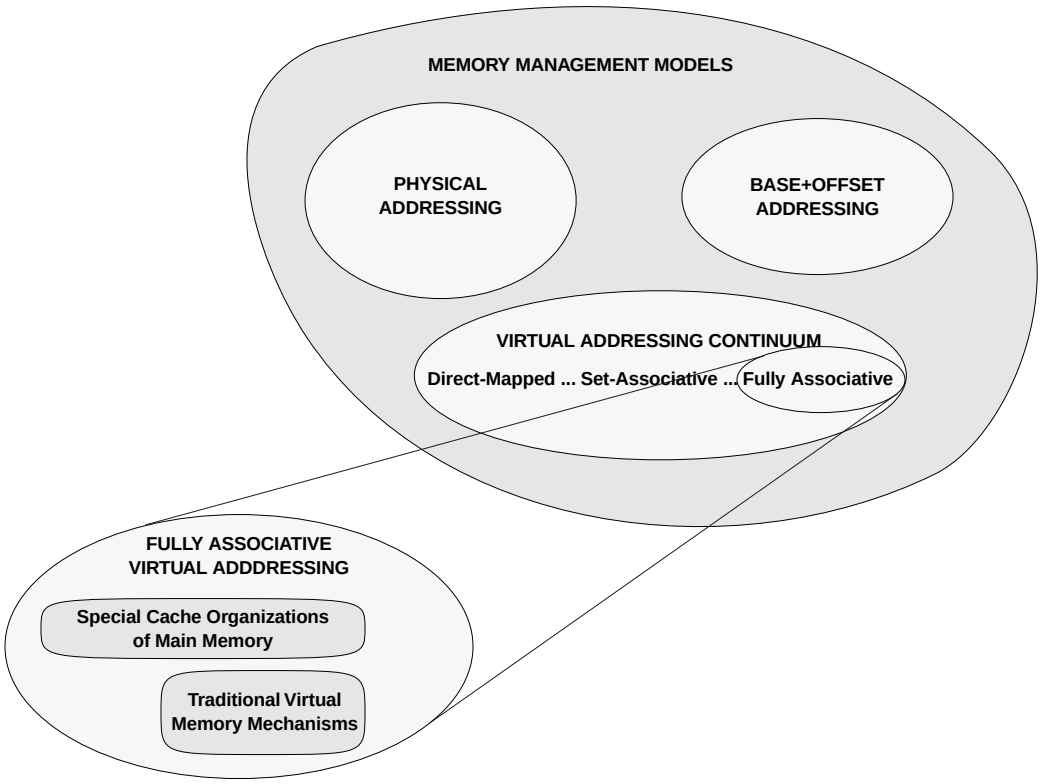

Figure 1: Memory management models

There are at least three ways to manage physical memory. The first, *physical addressing*, uses physical addresses in the program itself. The second, *base+offset addressing*, uses relative offsets in the program and adds the physical base address at runtime. The third, *virtual addressing*, uses any appropriate naming scheme in the program (usually relative offsets) and relies upon the operating system and hardware to translate the references to physical addresses at runtime. Traditional virtual memory is therefore a small subset of the virtual addressing model of memory management.

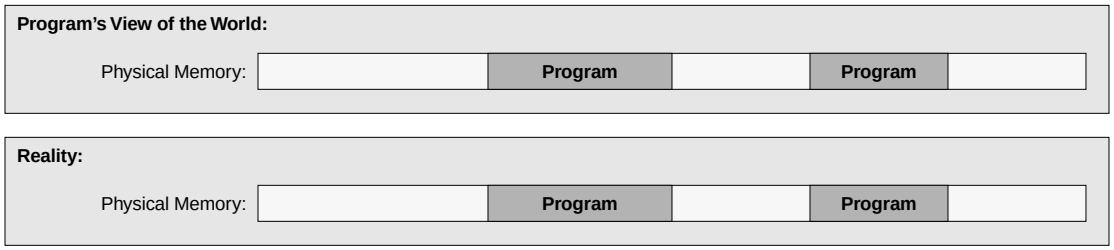

Figure 2: The Physical Addressing model of program creation and execution

In this model, a program's view of itself in its environment (physical memory) is equivalent to reality; a program contains knowledge of the structure of the hardware. A program can be entirely contiguous in memory, or it can be split into multiple pieces, but the locations of all parts of the program are known to the program's code and data.

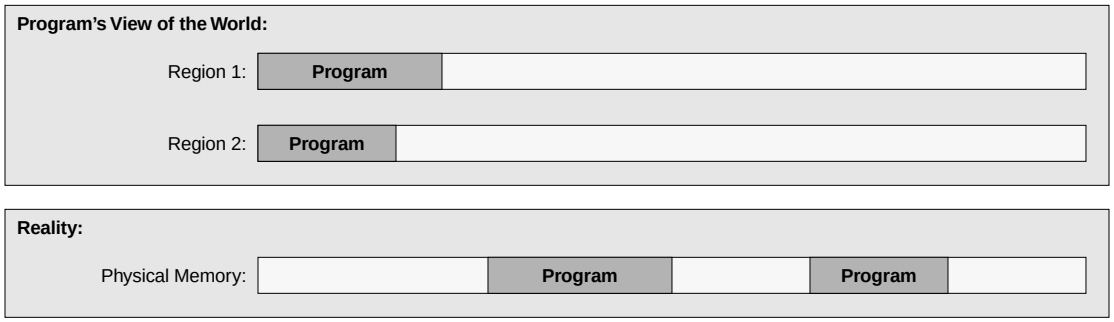

Figure 3: The Base+Offset Addressing model of program creation and execution

In this model, a program's view of itself in its environment is not equivalent to reality, but it sees itself as a set of contiguous regions. It does not know where in physical memory these regions are located, but the regions must be contiguous; they cannot be fragmented.

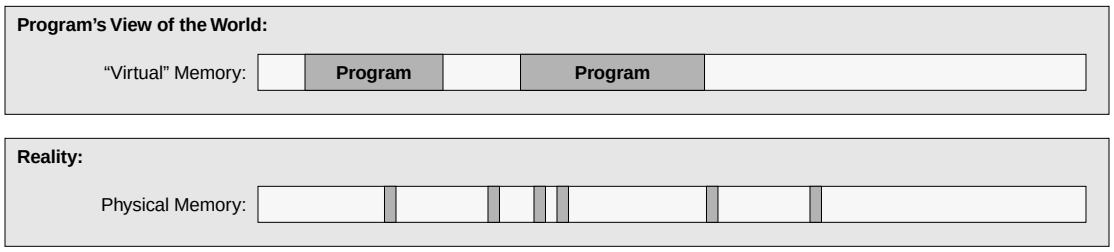

Figure 4: The Virtual Addressing model of program creation and execution

In this model, a program's view of itself in its environment has virtually nothing to do with reality; a program can consider itself a collection of contiguous regions or a set of fragments, or one large monolithic program. The operating system considers the program nothing more than a set of uniform virtual pages, and loads them as necessary into physical memory. The entire program need not be resident in memory, and it need not be contiguous.

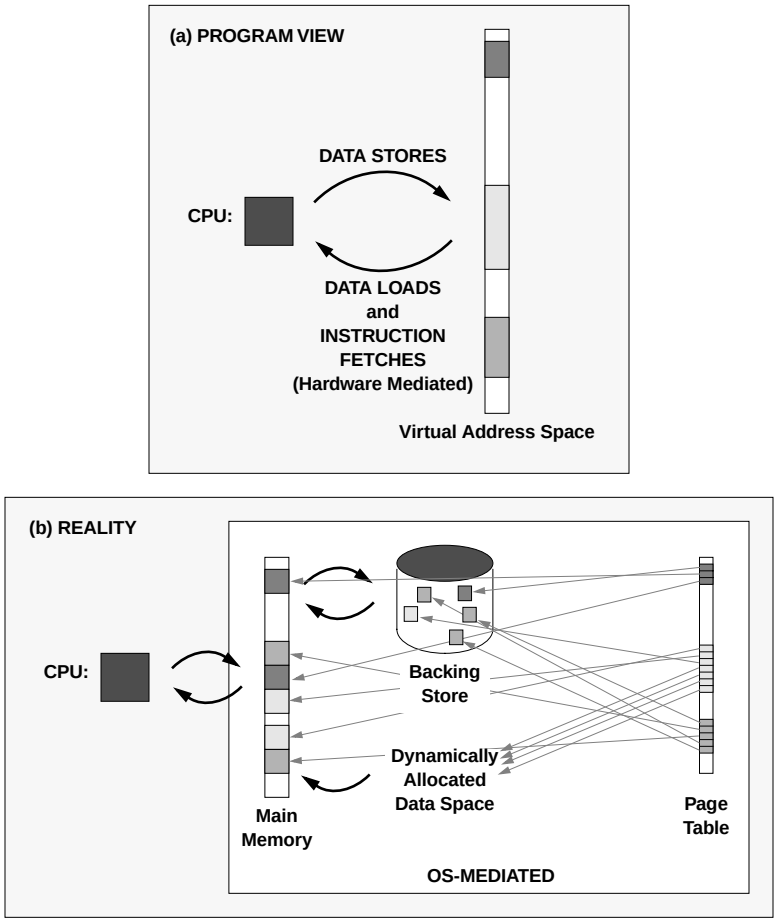

Figure 5: Caching the process address space in main memory

In the first view, a program is shown referencing locations in its virtual address space. All loads, stores, and fetches use virtual addresses to reference objects. The second view shows that the address space is not a linear object stored on some device, but is instead scattered across main memory and hard drives and even dynamically allocated when necessary. The page table handles the translation from virtual address space to physical location. Note that it has the same shape as the address space in figure (a), indicating that for every chunk of data in the virtual address space (called a “virtual page”), there is exactly one translation entry in the page table.