

Virtual Memory Management

Because main memory (*i.e.*, transistor memory) is much more expensive, per bit than disk memory (presently, approximately 10 to 50 times more expensive), it is usually economical to provide most of the memory requirements of a computer system as disk memory. Disk memory is also "permanent" and not (very) susceptible to such things as power failure. Data, and executable programs, are brought into memory, or *swapped* as they are needed by the CPU in much the same way as instructions and data are brought into the cache. Most large systems today implement this "memory management" using a hardware memory controller in combination with the operating system software.

One of the most primitive forms of "memory management" is often implemented on systems with a small amount of main memory. This method leaves the responsibility of memory management entirely to the programmer; if a program requires more memory than is available, the program must be broken up into separate, independent sections and one "overlaid" on top of another when that particular section is to be executed. This type of memory management, which is completely under the control of the programmer, is sometimes the only type of memory management available for small microcomputer systems. Modern memory management schemes, usually implemented in mini - to mainframe computers, employ an automatic, user transparent scheme, usually called "virtual memory".

Virtual memory is a computer design feature that permits [software](#) to use more [memory](#) than the computer physically possesses. In technical terms, it allows software to run in a memory address space whose size and addressing are not necessarily tied to the [computer's physical memory](#). It is often a feature of a computer's [operating system](#) and may or may not be a feature of the computer's hardware.

Explanation

Simply put, when a memory location is read or written to, [hardware](#) within the computer translates the memory address generated by the software (the *virtual memory address*) into a, usually distinct, real memory address (the *physical memory address*) within the computer's memory. This is accomplished by preserving the low order bits of the binary representation of the input address while treating the high order bits as a key to one or more address translation tables. For this reason a range of consecutive addresses in the virtual address space whose size is a [power of two](#) will be translated in a corresponding range of consecutive physical addresses. The memory referenced by such a range is called a *page*. The page size is typically in the range of 512 to 8192 bytes (with 4K being very common), though page sizes of 4 megabytes or larger may be used for special purposes. Using the same or a related mechanism, contiguous regions of virtual memory larger than a page are often mappable to contiguous physical memory for purposes other than virtualization, such as setting access and cache control bits.

The translation is implemented by an [MMU](#). This may be either a module of the [CPU](#) or an auxiliary, closely coupled chip. The MMU may have the ability to monitor page references according to the type of reference (for read, write or execution) and the privilege mode of the CPU at the time the reference was generated. In addition, the MMU may detect that a reference is to a page that is marked as unavailable. The MMU responds to such conditions by raising an [exception](#) with the CPU which will be trapped by system software. This makes it possible for an operating system to carry out **swapping**. This is the behaviour of copying one page of memory to disk in order to restore to memory another page copied to disk earlier. This can be realized in a swap partition, a dedicated section of disk in order to hold paged memory, or in a swap file.


To minimize the performance penalty of address translation, most modern CPUs include an on-chip MMU, and maintain a table of recently used physical-to-virtual translations, called a [Translation Lookaside Buffer](#), or TLB. Addresses with entries in the TLB require no additional time to translate.

The most fundamental advantage of virtual memory is that it allows a computer to multiplex its CPU and memory between multiple programs without the need to perform expensive copying of the programs' memory images. It also allows the operating system to protect its own code from corruption by an erroneous application program and to protect application programs from each other and (to some extent) from themselves. If the combination of virtual memory system and operating system supports swapping, then the computer may be able to run simultaneously programs whose total size exceeds the available physical memory. This is possible because most programs have a small subset (*active set*) of pages that they reference over significant periods their execution. If too many programs are run at once, then copies to and from disk will become excessively frequent and overall system performance will become unacceptably slow. This is often called *thrashing* (since the disk is being excessively overworked - thrashed) or *paging storm* and corresponds to access to the swap medium being three orders of magnitude or more slower than access to main memory.

Note that virtual memory is not a requirement for precompilation of software, even if the software is to be executed on a multiprogramming system. Precompiled software may be loaded by the operating system which has the opportunity to carry out address relocation at load time. This suffers by comparison with virtual memory in that a copy of program relocated at load time cannot run at a distinct address once it has started execution.

It is possible to avoid the overhead of address relocation using a process called [rebasing](#), which uses metadata in the executable image header to guarantee to the run-time loader that the image will only run within a certain virtual address space. This technique is used on the system libraries on Win32 platforms, for example.

In [embedded systems](#), swapping is typically not supported—virtual memory is primarily a convenience to cope with software errors

In a computer system which supports virtual memory management, the computer appears to the programmer to have its address space limited only by the addressing range of the computer, not by the amount of memory which is physically connected to the computer as main memory. In fact, *each process* appears to have available the full memory resources of the system. Processes can occupy the same virtual memory but be mapped into completely different physical memory locations. Of course, the parts of a program and data which are actually being executed must lie in main memory and there must be some way in which the "virtual address" is translated into the actual physical address in which the instructions and data are placed in main memory. The process of translating, or mapping, a virtual address into a physical address is called *virtual address translation*. Figure  shows the relationship between a named variable and its physical location in the system.

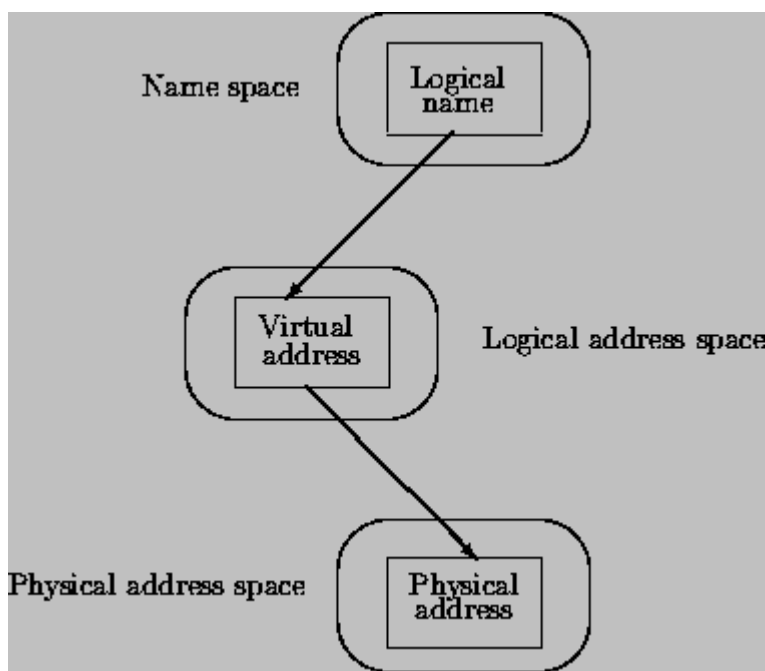


Figure: The name space to physical address mapping

This mapping can be accomplished in ways similar to those discussed for mapping main memory into the cache memory. In the case of virtual address mapping, however, the relative speed of main memory to disk memory (a factor of approximately 10,000 to 100,000) means that the cost of a "miss" in main memory is very high compared to a cache miss, so more elaborate replacement algorithms may be worthwhile.) In fact, in most processors, a direct mapping scheme is supported by the system hardware, in which a *page map* is

maintained in physical memory. This means that each physical memory reference requires *both* an access to the page table *and* an operand fetch. In effect, all memory references are indirect. Figure 1 shows a typical virtual-to-physical address mapping.

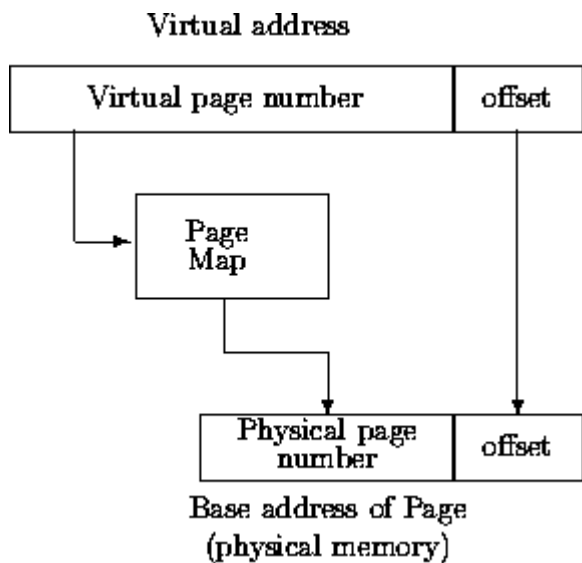


Figure: A direct mapped virtual to physical address translation

This requirement would be a considerable performance penalty, so most systems which support virtual addressing have a small associative memory (called a *translation lookaside buffer*, or TLB) which contains the last few virtual addresses and their corresponding physical addresses, so in most cases the virtual to physical mapping does not require an additional memory access. Figure 2 shows a typical virtual-to-physical address mapping in a system containing a TLB.

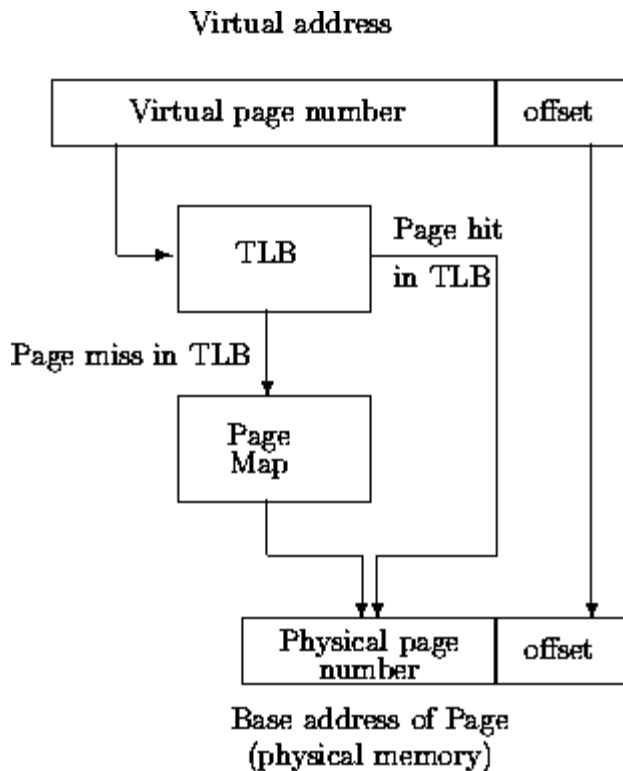


Figure: A virtual to physical address translation mechanism with a TLB

For many current architectures, including the VAX, INTEL 80486, and MIPS, addresses are 32 bits, so the virtual address space is 2^{32} bytes, or 4 G bytes. A physical memory of about 16-64 M bytes is typical for these machines, so the virtual address translation must map the 32 bits of the virtual memory address into a corresponding area of physical memory.

Sections of programs and data not currently being executed normally are stored in disk, and are brought into main memory as necessary. If a virtual memory reference occurs to a location not currently in physical memory, the execution of that instruction is aborted, and can be restored again when the required information is placed in main memory from the disk by the memory controller. (Note that, when the instruction is aborted, the processor must be left in the same state it would have been had the instruction not been executed at all). While the memory controller is fetching the required information from disk, the processor can be executing another program, so the actual time required to find the information on the disk (the disk seek time) is not wasted by the processor. In this sense, the disk seek time usually imposes little (time) overhead on the computation, but the time required to actually place the information in memory may impact the time the user must wait for a result. If many disk seeks are required in a short time, however, the processor may have to wait for information from the disk.

Normally, blocks of information are taken from the disk and placed in the memory of the processor. The two most common ways of determining the sizes of the blocks to be moved into and out of memory are

called *segmentation* and *paging*, and the term *segmented memory management* or *paged memory management* refer to memory management systems in which the blocks in memory are *segments* or *pages*.

Segmented memory management

In a segmented memory management system the blocks to be replaced in main memory are potentially of unequal length and correspond to program and data "segments." A program segment might be, for example, a subroutine or procedure. A data segment might be a data structure or an array. In both cases, segments correspond to logical blocks of code or data. Segments, then, are "atomic," in the sense that either the whole segment should be in main memory, or none of the segment should be there. The segments may be placed anywhere in main memory, but the instructions or data in one segment should be contiguous, as shown in Figure 10.1.

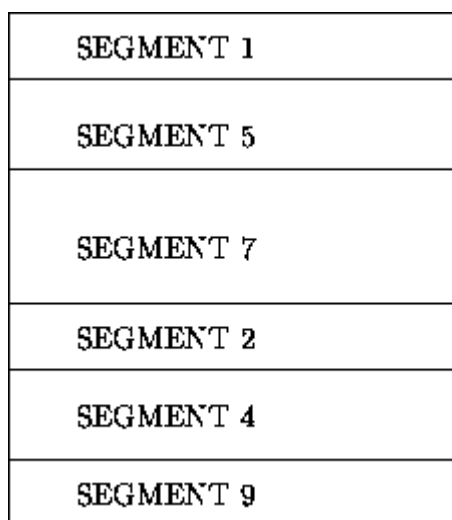


Figure: A segmented memory organization

Using segmented memory management, the memory controller needs to know where in physical memory is the start and the end of each segment. When segments are replaced, a single segment can only be replaced by a segment of the same size, or by a smaller segment. After a time this results in a "memory fragmentation", with many small segments residing in memory, having small gaps between them. Because the probability that two adjacent segments can be replaced simultaneously is quite low, large segments may not get a chance to be placed in memory very often. In systems with segmented memory management, segments are often "pushed together" occasionally to limit the amount of fragmentation and allow large segments to be loaded.

This organization appears to be efficient because an entire block of code is available to the processor. Also, it is easy for two processes to share the same code in a segmented memory system; if the same procedure is

used by two processes concurrently, there need only be a single copy of the code segment in memory. (Each process would maintain its own, distinct data segment for the code to access, however.)

Segmented memory management is not as popular as paged memory management, however. In fact, most processors which presently claim to support segmented memory management actually support a hybrid of paged and segmented memory management, where the segments consist of multiples of fixed size blocks.


Paged memory management:

Paged memory management is really a special case of segmented memory management. In the case of paged memory management,

- all of the segments are exactly the same size (typically 256 bytes to 16 K bytes)
- Virtual "pages" in auxiliary storage (disk) are mapped into fixed page-sized blocks of main memory with predetermined page boundaries.
- The pages do not necessarily correspond to complete functional blocks or data elements, as is the case with segmented memory management.

The pages are not necessarily stored in contiguous memory locations, and therefore every time a memory reference occurs to a page which is not the page previously referred to, the physical address of the new page in main memory must be determined. In fact, most paged memory management systems (and segmented memory management systems as well) maintain a "page translation table" using associative memory to allow a fast determination of the physical address in main memory corresponding to a particular virtual address. Normally, if the required page is not found in the main memory (i.e, a "page fault" occurs) then the CPU is interrupted, the required page is requested from the disk controller, and execution is started on another process.

The following is an example of a paged memory management configuration using a fully associative page translation table:

Consider a computer system which has 16 M bytes (2^{24} bytes) of main memory, and a virtual memory space of 2^{32} bytes. Figure  shows a sketch of the page translation table required to manage all of main memory if the page size is $4K^{(2^{12})}$ bytes. Note that the associative memory is 20 bits wide (32 bits - 12 bits, the virtual address size -- the page size). Also to manage 16 M bytes of memory with a page size of 4 K bytes, a total of $(16M = 2^{24}) / (4K = 2^{12}) = 2^{12} = 4096$ associative memory locations are required.

not be very full, since the number of entries is limited to the amount of physical memory available. One way these large, sparse PTT's are managed is by mapping the PTT itself into virtual memory. (Of course, the pages which map the virtual PTT must not be mapped out of the physical memory!)

Note that both paged and segmented memory management provide the users of a computer system with all the advantages of a large virtual address space. The principal advantage of the paged memory management system over the segmented memory management system is that the memory controller required to implement a paged memory management system is considerably simpler. Also, the paged memory management does not suffer from fragmentation in the same way as segmented memory management. Another kind of fragmentation does occur, however. A whole page is swapped in or out of memory, even if it is not full of data or instructions. Here the fragmentation is within a page, and it does not persist in the main memory when new pages are swapped in.

One problem found in virtual memory systems, particularly paged memory systems, is that when there are a large number of processes executing ``simultaneously" as in a multi-user system, the main memory may contain only a few pages for each process, and all processes may have only enough code and data in main memory to execute for a very short time before a page fault occurs. This situation, often called ``thrashing," severely degrades the throughput of the processor because it actually must spend time waiting for information to be read from or written to the disk.