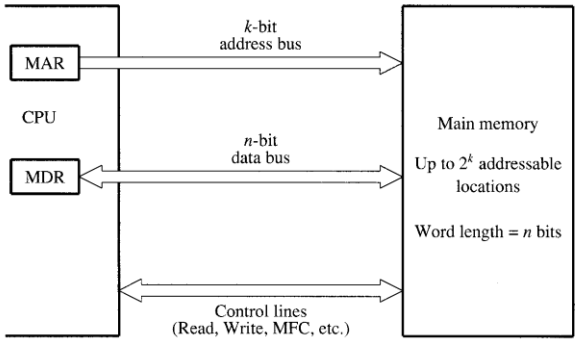
THE MAIN MEMORY

Programs and the data they operate on are held in the ***main memory*** (MM) of the computer during execution. We really should discuss the way in which this vital part of the computer operates.

By now, you appreciate the fact that the execution speed of instructions is highly dependent upon the speed with which ***instructions and data*** can be transferred to or from the MM. Thus it is not surprising that memory design is a very important topic in computer development.



In most modern computer systems, the physical MM ***is not as large*** as the address space spanned by an address issued by the processor. The diagram shows that the CPU may send addresses which are ***k*** bits long – therefore, the promise of the MM is that it will accept and service any address in the range:
0 to 2^k -1.

When programs do not totally fit into the MM, the parts of it not currently being executed are stored on secondary storage devices such as mechanical magnetic hard drives. Of course, all parts of a program that are eventually executed are first brought into the MM. When a new segment (page or frame) of a program is to be moved into a full MM, it must possibly replace another segment (page or frame) already in the MM. Modern computers can manage such operations – page swapping -- automatically, so that the programmer need not be concerned with their details.

SOME BASIC CONCEPTS

The maximum size of the MM that can be used in any computer is determined by the addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing up to 2^{16} (64K) memory locations. Similarly, a machine whose instructions generate 32-bit addresses can utilize an MM that contains up to 2^{32} (4G) memory locations. This number represents the size of the ***virtual address space*** of the computer.

In some computers, the smallest addressable unit of information is a single memory byte (*n=8*), whereas in others, memory is word-addressable (*n=16 or n=32*).

So, either individual memory bytes may be assigned distinct addresses, yielding a byte-addressable computer, or a memory word contains one or more memory bytes that can be addressed individually. For example, a byte-addressable 32-bit computer, each memory word contains 4 bytes. The figure shows possible address assignment in this case.

Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11

The address of a word is that of its first-order byte; thus, word addresses are always integer multiples of 4. (*n=32*).

The MM is usually designed to store and retrieve data in word-length quantities, fact, the number of bits actually stored or retrieved in one MM access is the most common designation of the word length of a computer. Consider, for example, a byte addressable computer with the addressing structure of the above figure, whose instructions generate 32-bit addresses. When a 32-bit address is sent from the CPU to the MM unit, the high-order 30 bits determine which ***word*** will be accessed. If a byte quantity is specified, the low-order 2 bits of the address specify which ***byte location*** is involved. So, in the case of a Read operation, other bytes may be fetched from the MM, but they are stored or ignored by the CPU.

If the byte operation is a Write, however, the control circuitry the MM must ensure that the contents of other bytes of the same word are not changed.

From the system standpoint, we can view the MM unit as a "black box." Data transfer between the MM and the CPU takes place through the use of two CPU sisters, usually called MAR (memory address register) and MDR (memory data sister

register). If the MAR is ***k*** bits long and MDR is ***n*** bits long, then the MM unit may store up to 2^k addressable locations. And, during a "memory cycle," ***n*** bits of data transferred between the MM and the CPU. This transfer takes place over the processor bus, which has ***k*** address lines and ***n*** data lines.

The bus also includes the control lines Read, Write, and WMFC plus Memory Function Completed (MFC) for coordinating a transfer. In the case of byte-addressable computers, another control line may be added to indicate when only a byte, rather than a full word of n bits, is to be transferred The connection between the CPU and the MM is shown schematically the first figure.

The CPU initiates a memory operation by storing the appropriate data into registers MDR and MAR, then setting either the Read or Write memory control line to 1 and raising the WMFC signal, indicating that it is awaiting a response. When the required operation is completed, memory control circuitry indicates this to the CPU by setting MFC to 1. WMFC and MFC are "handshaking signals".

A useful measure of the speed of memory units is the time that elapses between initiation of an operation and the completion of that operation (for example, the time between a Read request, and receipt of the MFC). This is referred to as the memory access time, averaged over many cycles.

Another important measure is the memory cycle time, which is the minimum time delay required between the initiation of two successive memory operations (for example, the time between two successive Read operations). The cycle time is usually slightly longer than the access time, depending upon the implementation details of the memory unit.

Recall that a memory unit is called a random-access memory (RAM) if any location can be accessed for a Read or Write operation in some fixed amount of time that is independent of the location's address. Main memory units are of this type. This distinguishes them from serial, or partly serial, access storage devices such as magnetic disks. Access time on the latter devices depends upon the address or position of the data. (Statistically, on an old tape back-up system, the required information will be in the middle of the tape).

The basic technology for the implementation of main memories uses semiconductor integrated circuits. We will then consider some of the techniques used to increase the effective speed and size of the main memory.

The CPU of a computer can usually process instructions and data faster than they can be fetched from a ***compatibly priced main memory unit***. The memory cycle time, then, is the bottleneck in the system. One way to reduce the memory access time is to use a cache memory. This is a small and fast memory that is inserted between the larger, slower main memory and the CPU. It holds the currently active segments of a program and its data. Another technique, called ***memory interleaving***, divides the system into a number of memory modules and, importantly, arranges addressing so that successive words in the address space are placed in different modules. Since requests for memory access tend to involve consecutive addresses, as when executing straight-line program segments, then the accesses will be to different modules. Since parallel access to these modules is possible, the average rate of fetching words from the MM can be increased.

Virtual memory is another important concept related to memory organization. So far, we have assumed that the addresses generated by the CPU directly specify specific locations in the MM. This may not always be the case. For reasons that will become apparent, data or instructions may be stored in physical memory (PM) locations that have addresses different from those specified by the program – virtual memory addresses or addresses in VM. The memory control circuitry ***translates*** the address specified by the program – ***a virtual memory*** address into a ***physical memory address*** that can be used to access the ***real physical memory***. In such a case, an address generated by the CPU is referred to as a virtual or logical address. The virtual address space is mapped onto the physical memory where data is actually stored. The mapping function is implemented using special memory control circuits, often called the memory management unit (MMU). This mapping function may be changed during program execution according to system requirements, and uses a key component known as the ***page table.***

Virtual memory can be used to increase the effective size of the MM. Data are accessed in a virtual address space that can be as large as the addressing capability the CPU, but**, at any given time, only the active portion of this space is mapped locations in the physical main memory.** The remaining virtual addresses are mapped onto the bulk storage devices used.

As the active portion of the virtual address space changes during program execution, the memory management unit changes the mapping function by updating the page table, and transfers data between the bulk storage and the main memory. Thus, during every memory cycle, an address-processing mechanism (hardware or software) determines whether the addressed information is in the physical DRAM MM unit. Thus, then the proper word is accessed and execution proceeds. If it is not, a contiguous page

of words containing the desired word is transferred from the bulk storage to the MM, possibly displacing some page that is currently determined to be relatively inactive.

Because of the required for movement of pages between bulk storage and the MM, there is a degradation in this type of a system. By judiciously choosing which pages to be in the MM, however, there may be reasonably long periods during which the ability is high that the words accessed by the CPU are in the physical MM unit.

This section has briefly introduced a number of organizational features of memory systems. These features have been developed to help provide a computer system as large and as fast an MM component as can be afforded in relation to the overall of the system.

SEMICONDUCTOR RAM MEMORIES

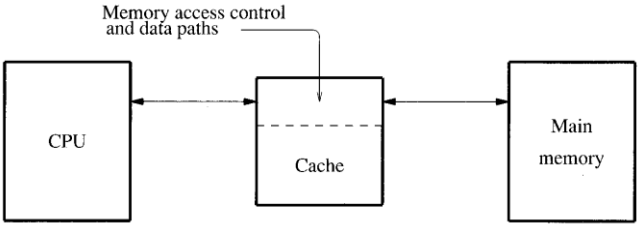
Semiconductor memories are available in a wide range of speeds. Their cycle times from a few hundred nanoseconds to a few nanoseconds. Historically, when first introduced in the late 1960s, they were much more expensive than the magnetic-core memories they replaced. But, because of the rapid advances in VLSI (very large scale integration) technology, however, the cost of semiconductor memories has dropped dramatically. As a result, they are now used exclusively in the implementation of main memories.

CACHE MEMORIES

Analysis of a large number of typical programs has shown that most of their execution time is spent on a few main routines in which a number of instructions are executed repeatedly. Also, data, such as arrays or strings, have elements which are stored at addresses where one follows the other.

Instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other. The actual detailed pattern of instruction sequencing is not important. The main observation is that many instructions in a few localized areas of the program are repeatedly executed and that the remainder of the program is accessed relatively infrequently. This phenomenon is referred to as ***locality of reference***, and also applied to data. ***Locality of reference*** is the reason why we can use a paging mechanism, and also makes caching possible. It, and the tendency for programs to be written in a ***Straight Line Sequence*** make a memory hierarchy consisting of the hard-drive (VM), the DRAM and the cache a viable high-speed system. Without these two phenomena, we would have to redesign our memory architecture completely.

If the active segments of a program can be placed in a ***fast memory***, then the total execution time can be significantly reduced. Such a memory is referred to as a cache memory, which is inserted between the CPU and the main memory as shown. To make this arrangement effective, the cache must be considerably faster than the MM. This approach is more economical than the use of fast memory devices to implement the entire MM.



Conceptually, operation of a cache memory is very simple. The memory control circuitry is designed to take advantage of the property of locality of reference.

When a Read request is received from the CPU, the contents of a ***block*** (NOT A PAGE!) of memory words containing the location specified are transferred into the cache one word at a time. When any of the locations in this block is referenced by the program, its contents are read directly from the cache. Usually, the cache memory can store a number of such blocks at any given time. The correspondence between the MM blocks and those in the cache is specified by a ***mapping function***. Depending upon the mapping function, sometimes, when the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the replacement algorithm, and similar decisions must be made about the replacement of ***pages*** in the DRAM MM.

The CPU does not need to know explicitly about the existence of the cache. It simply makes Read and Write requests using addresses that refer to locations in the MM. The memory-access control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. When the operation is a Read, the main memory is not involved. If the operation is a Write, the situation is more complex, as you must

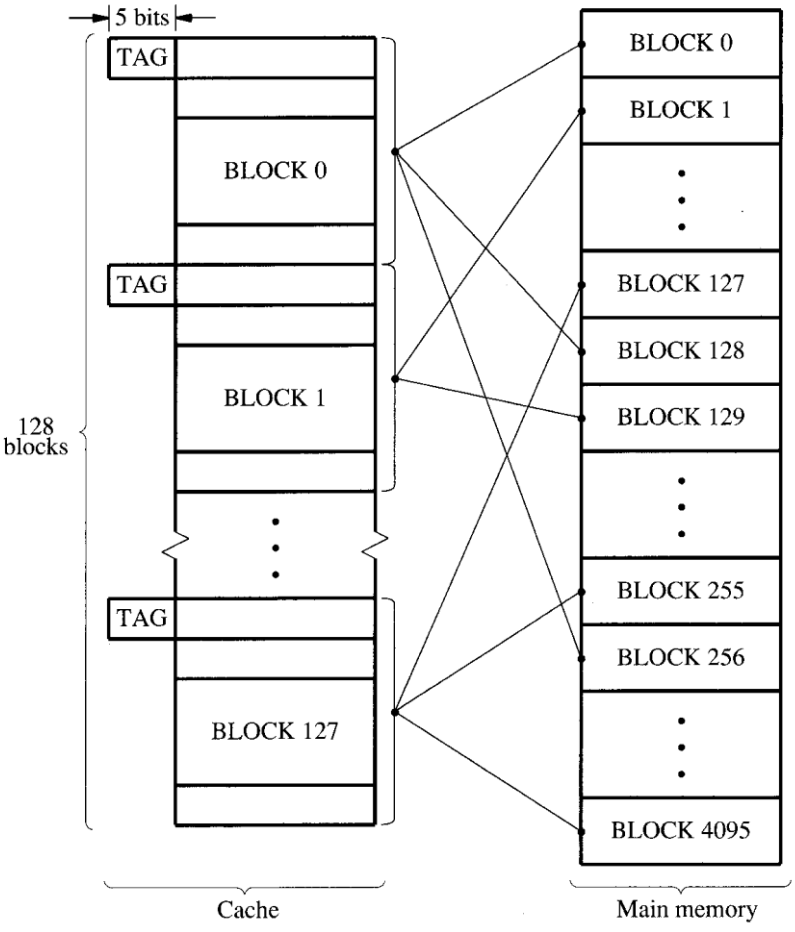
preserve any changes which are made. So, the system can proceed in two ways. (1) In the first technique, called ***store-through***, the cache location and the MM location are updated simultaneously. (2) The alternative is to update only the cache location and to ***mark it*** as updated with an associated flag bit, often called the ***dirty*** or ***modified bit***. The permanent MM location of the word is updated later, when the block containing this marked word is to be removed from the cache to make way for a new block. The store-through method is clearly simpler, but it results in unnecessary Write operations in the MM when a given cache word is updated a number of times during its cache residency period.

When the addressed word in a Read operation is not in the cache, the block of words that contains the requested word is copied from the MM into the cache. Then, the particular word requested is forwarded to the CPU. The particular word that was requested may be forwarded to the CPU after the entire block is loaded into the cache. Alternatively, this word may be sent to the CPU as soon as it is read from the main memory. The latter approach, which is called load-through, reduces the CPU's waiting period somewhat. However, the resulting improvement in performance is often not worth the associated complexity, particularly in small to medium size computers.

During a Write operation, if the addressed word is not in the cache, then the information is written directly into the MM. In this case, there is little advantage in transferring the block containing the addressed word to the cache. A Write operation normally refers to a location in one of the data areas of a program rather than to the memory area containing program instructions. The property of locality of reference is not as pronounced in accessing data when Write operations are involved.

Finally, we should recall that in the case of an interleaved memory, contiguous block transfers are very efficient. Transferring data in blocks between the MM and the cache enables an interleaved MM unit to operate at its maximum possible speed. Similarly, block transfers to the cache can take advantage of the block or burst transfer mode available on most dynamic RAM chips.

Mapping Functions

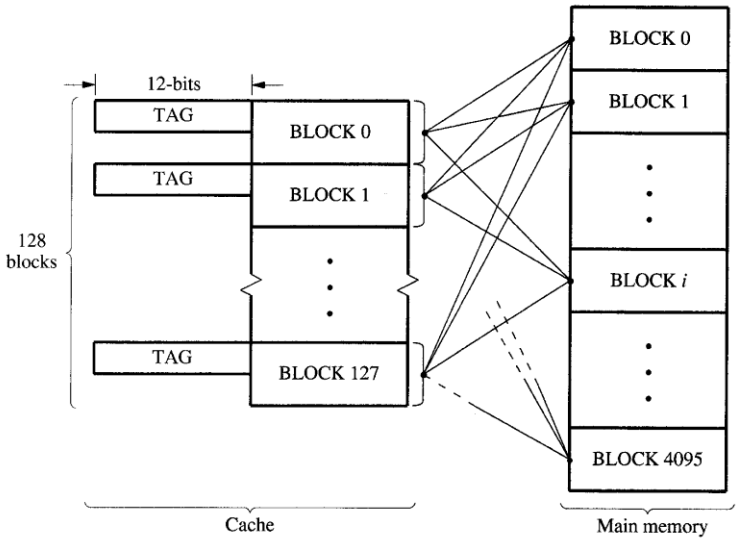


In order to discuss possible methods for specifying where MM blocks are placed in the cache, we will use a specific example. Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by a 16-bit address. For mapping purposes, the main memory will be viewed as composed of 4K blocks of 16 words.

The simplest way to associate MM blocks with cache blocks is the ***direct-mapping*** technique. In this technique, block *k* of the MM maps onto block *k modulo 128* of the cache, as depicted in the figure. Since more than one MM block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full. For example, instructions of a program may start in block 128 and continue in block 256, possibly after a branch. As this program is executed, both of these blocks must be transferred to the cache-block-0 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block. In this case, the replacement algorithm is trivial.

The tag field of that block is compared to the appropriate bits in the address. If they match, then the desired word is in that block of the cache; This is a cache “hit”. If there is no match, then the block containing the required word must first be read from the main

memory and loaded into the cache. That is a "cache miss". The direct-mapping technique is easy to implement, but it is not very flexible.



This figure shows a much more flexible mapping method in which an MM block can potentially reside in any cache block position. In this case, 12 tag bits are required to identify an MM block when it is resident in the cache. The tag bits of an address received from the CPU are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the associative-mapping technique. Because of the complete freedom this technique gives in block positioning, a wide range of replacement algorithms is possible. However, its cost of implementation is somewhat higher than the cost of the direct-mapping scheme because of the need to search all 128 tag patterns to determine whether a given block is in the cache. A search of this kind is called an associative search. ***That's not good!!***

The final mapping method to be discussed is a combination of the two techniques mentioned above. Blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this block-set-associative-mapping technique would be for a cache with two blocks per set. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

We should note that the number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. Four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping method. (See the last two diagrams for these extremes).

Another technical detail that should be mentioned, is the fact that a ***valid*** bit must usually be provided for each block. This bit indicates whether the block contains valid data. It should not be confused with the ***modified or dirty*** bit mentioned earlier. The dirty bit indicates whether the block has been modified during its cache residency and is needed only in systems that do not use the store-through method. The valid bits are all set to 0 when power is initially applied to the system or when the MM is loaded with new programs and data from mass storage devices.

Transfers from mass storage devices are carried out by a DMA mechanism, and normally bypass the cache. The valid bit of a particular cache block is set to I the first time this block is loaded from the MM, and it stays at I unless an MM block is updated by a source that bypasses the cache. In this case, a check is made to determine whether the block is currently in the cache. If it is, its valid bit is set to 0.

Replacement Algorithms

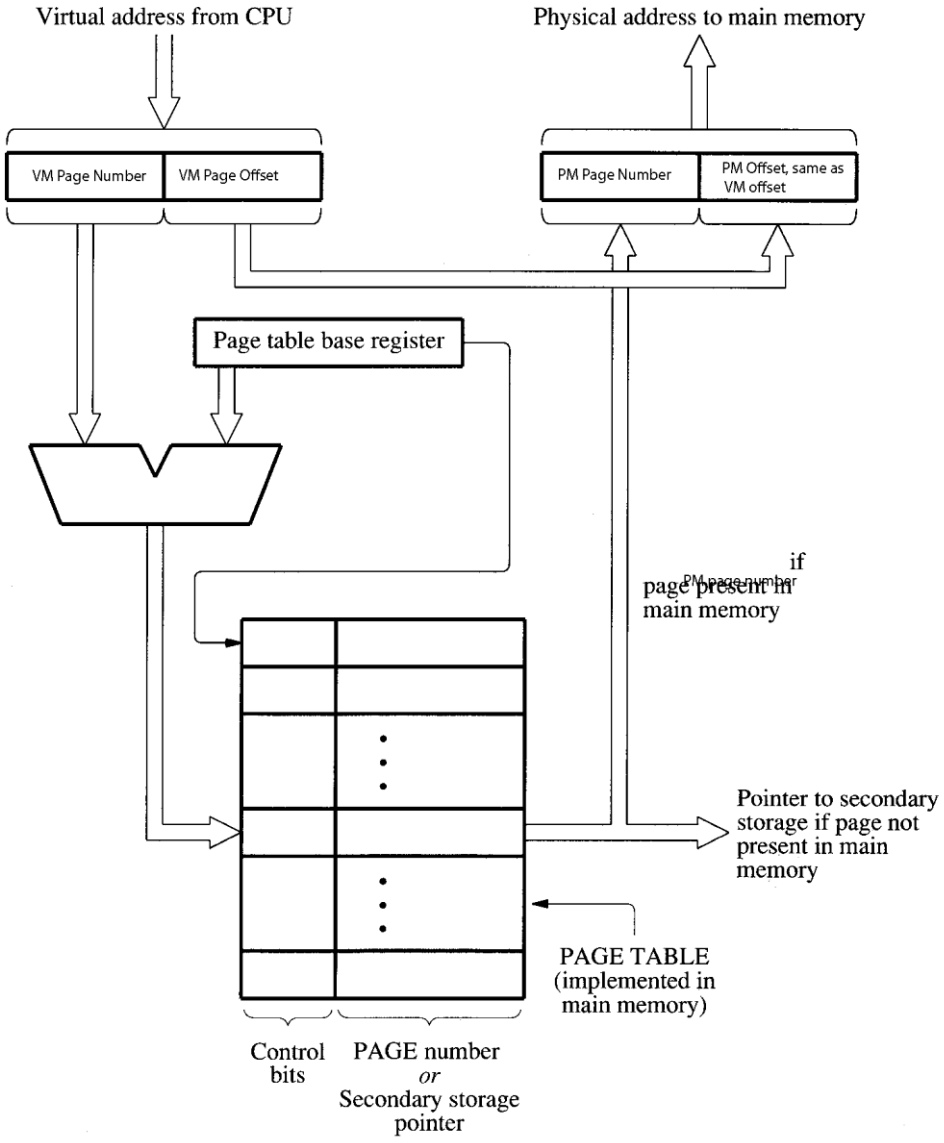
When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This problem has generated a great deal of interest among computer scientists, because the decision can potentially be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. However, it is not easy to determine which blocks are about to be referenced. The property of ***locality of reference*** in programs gives a clue to a reasonable strategy. Because programs usually stay in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the ***longest time*** without being referenced. This block is called the ***least-recently-used (LRU)*** block, and the technique is called the LRU replacement algorithm.

In order to use the LRU algorithm, the cache controller must track the LRU block as computation proceeds. As a specific example, suppose it is required to track the LRU block of a four-block set. A 2-bit counter may be used for each block. When a hit occurs (when an access request is received for a word that is in the cache), the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged. When a miss occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one. When a miss occurs and the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0. The other three block counters are incremented by one. It can be easily verified that the counter values of occupied blocks are always distinct.

There are several replacement algorithms that require less overhead than the LRU approach. An intuitively reasonable rule would be to remove the "oldest" block from a full set when a new block must be brought in. Using this technique, no updating is needed when hits occur. However, because the algorithm does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU algorithm in choosing the best blocks to remove. The simplest algorithm is to choose the block to be overwritten at random. Interestingly enough, this simple algorithm has been found to be quite effective in practice.

VIRTUAL MEMORIES

In any computer system in which the currently active programs and data do not fit into the physical MM space, secondary storage devices such as magnetic disks hold the overflow. The operating system automatically moves instructions and data between the main memory and secondary storage. Thus, the application programmer does not need to be aware of the limitations imposed by the available main memory.



Techniques that automatically move program and data pages (a.k.a. frames) into the physical MM when they are required for execution are called virtual-memory techniques. Programs, and hence the processor, reference an instruction and data space that is independent of the available physical MM space. The binary addresses that the processor issues for either instructions or data are called ***virtual or logical addresses***. The mechanism that translates these into physical addresses is usually implemented by a combination of hardware and software components. If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. On the other hand, if the referenced address is not in the

MM, its contents must be brought into a suitable location in the MM before they can be used.

The Page Table base register's contents are *added* to the virtual memory address to permit *the location of the page table anywhere in memory*. If zero, the page table begins at address zero, but you can add a constant to puch it down the memory to any location.

The simplest method of translation assumes that all programs and data are composed of fixed-length units called ***pages***, each of which consists of a block of words that occupy contiguous locations in the MM or in secondary storage. Pages commonly range from 1K to 8K bytes in length. They constitute the basic unit of information that is moved back and forth between the MM and secondary storage whenever the translation mechanism determines that a move is required. This discussion clearly parallels many ideas that were introduced in the cache memory section.

The cache is intended to bridge the speed gap between the processor and the MM and is implemented in hardware. The virtual-memory idea, on the other hand, is meant to bridge the size gap between the MM and secondary storage and is usually implemented in part by software techniques. Conceptually, cache techniques and virtual-memory techniques are very similar. They differ mainly in the details of their implementation.

A virtual memory address translation method based on the concept of fixed-length pages is shown schematically in the figure. Each virtual address generated by the processor, whether it is for an instruction fetch or an operand fetch/store operation, is interpreted as a page number (high-order bits) followed by a word number (low-order bits). Information about the disk or main memory location of each page is kept in a page table in the main memory. The starting address of this table is kept in a page table base register. By adding the page number to the contents of this register, the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the MM; otherwise, they indicate where the page is to be found in secondary storage. In this case, the entry in the table usually points to an area in the main memory where the secondary storage address of the page is held. Each entry also includes some control bits to describe the status of the page while it is in the MM. One control bit indicates whether the page has been modified during its residency in the MM. As with cache memories, this information is needed to determine whether to write the page back to secondary storage before removing it from the MM to make room for another page.

Other control information may also be recorded in the control bits.

If the page table is stored in the MM unit, as we assumed above, then two MM accesses must be made for every MM access requested by a program. This would result in a degradation of execution speed by a factor of two. However, a specialized cache memory is used in most systems to speed up the translation process by storing recently used virtual to physical address translations.

When a program generates an access request to a page that is not in the MM, a page fault is said to have occurred. The whole page must be brought from secondary storage into the MM, usually via DMA, before access can proceed. Because a long delay is incurred while the page transfer takes place, the processor may execute another task whose pages are in the MM. It is the responsibility of the operating system to suspend execution of the program that caused the page fault and to start the execution of another program.

Once the MM is full, if a new page is brought from secondary storage, it must replace one of the resident pages. The problem of choosing the page to be removed is just as critical here as it is in a cache, and the notion that programs spend most of their time in a few localized areas is also applicable. Because main memories are considerably larger than cache memories, it should be possible to keep relatively larger portions of a program in the MM. This will reduce the frequency of transfers to and from secondary storage. Concepts similar to the LRU replacement algorithm can be applied to page replacement, and the required usage data can be kept in control bits in the page table entries.

MEMORY MANAGEMENT REQUIREMENTS

In our discussion of virtual-memory concepts, we have tacitly assumed that only one large program is being executed. If all of it does not fit into the available physical MM, parts of it (pages) are moved from secondary storage into the MM when they are to be executed and displace pages that have become idle. Although we have alluded to software routines that are needed to manage this movement of program segments, we have not been specific about the details. Of course, these management routines must also reside in the MM when they are executed.

The management routines are part of the operating system of the computer. It is convenient to assemble the operating system routines into a virtual address space, called the system space, that is separate from the virtual space in which user application programs reside. The latter space is called the user space. In fact, there may be a number of user spaces, one for each user. This can be arranged by providing a separate page table for each user. The particular table that is selected depends on the contents of the page table base register. By changing the contents of this register, the operating system can switch from one space to another. The physical MM is thus shared by the active pages of the system space and each of the user spaces. However, only the pages that belong to one of these spaces are accessible at any given time.

In any computer system in which independent user programs coexist in the MM, the notion of protection must be addressed. No program should be allowed to destroy either data or instructions of other programs in the MM. There are a number of ways in which such protection may be provided. Let us first consider the most basic form of protection. We introduce the notion of the ***state*** of the processor. In the simplest case there are ***two states***, the supervisor state and the user state. As the names suggest, the processor is usually placed in the supervisor state when operating system routines are being executed, and in the user state to execute user programs. In the user state, some machine instructions cannot be executed. These privileged instructions, which include operations such as modification of the page table base register, can only be executed while the processor is in the supervisor state. Hence, a user program is prevented from accessing the page table entries of other user spaces or of the system space.

It is often desirable for one application program to have access to pages belonging to another program. (Shared pages, like Window's .dll's). The operating system can arrange this by causing these pages to appear as shared spaces. The shared pages will therefore have entries which contain this information – shared or common libraries – in the page table. Thus, the control bits in each table entry can be set to control the access privileges granted to each program. For example, one program may be allowed to read and write a given page, while the other program may be given only read access.