Inside the Cache Manager

Mark Russinovich

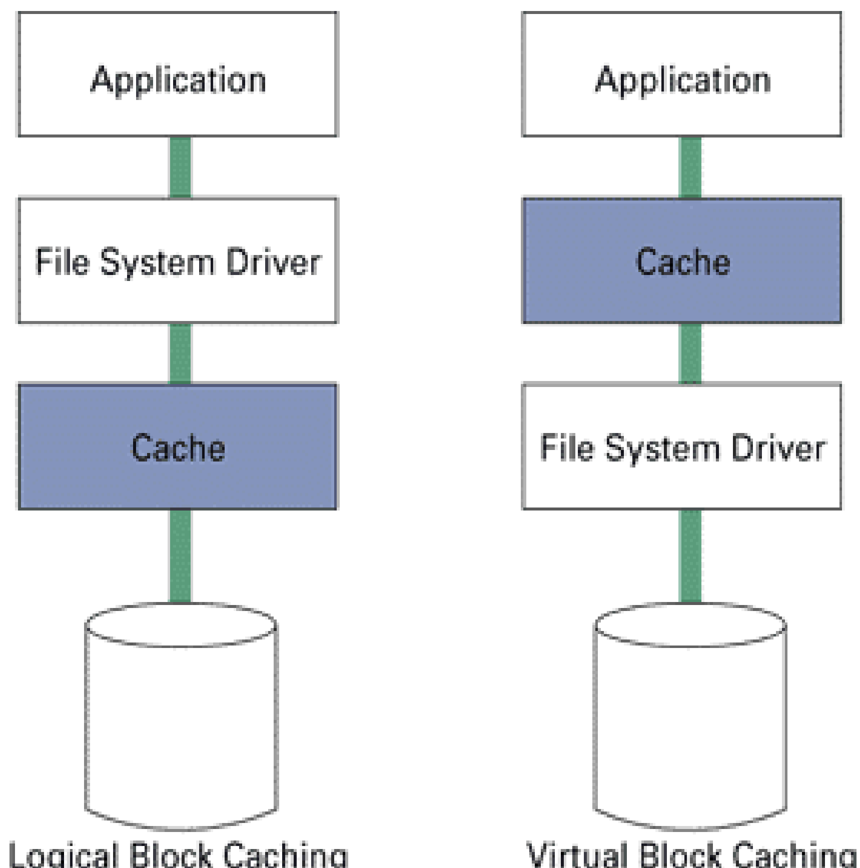
(Reprinted from WindowsItPro Magazine)

Caching file system data is an important performance optimization that virtually every modern operating system (OS) performs. The premise behind caching is that most applications access data that is primarily localized within a few files. Bringing those files into memory and keeping them there for the duration of the application's accesses minimizes the number of disk reads and writes the system must perform. Without caching, applications require relatively expensive disk operations every time they access a file's data.

This month, I'll look inside the Cache Manager, an Executive subsystem that works closely with the Memory Manager and file systems to cache file-system data in memory. In my last two columns about the Memory Manager, you can find details about many of the concepts I refer to this month (such as mapping a view of a file).

Logical Block and Virtual Block Caching

OSs use two types of file-system data caching: logical block caching and virtual block caching. The two types store data at different levels of abstraction, as Figure 1, page 68 shows. A logical drive resides on a disk partition that's composed of physical storage units called sectors. When an application accesses data in a particular file, the file system responsible for the drive (e.g., FAT, NTFS) determines which sectors of the disk hold the data in the file. The file system then issues disk I/O requests to read from or write to those sectors. In logical block caching, the OS caches sector data in memory so that the memory associated with the target sectors, rather require disk operations, can satisfy disk I/O requests. Most older variants of the UNIX OS, including BSD 4.3, every Microsoft OS (Windows 98, Win95, Windows 3.x, and DOS) except Windows NT, and Novell NetWare, cache file system data at the logical block level.

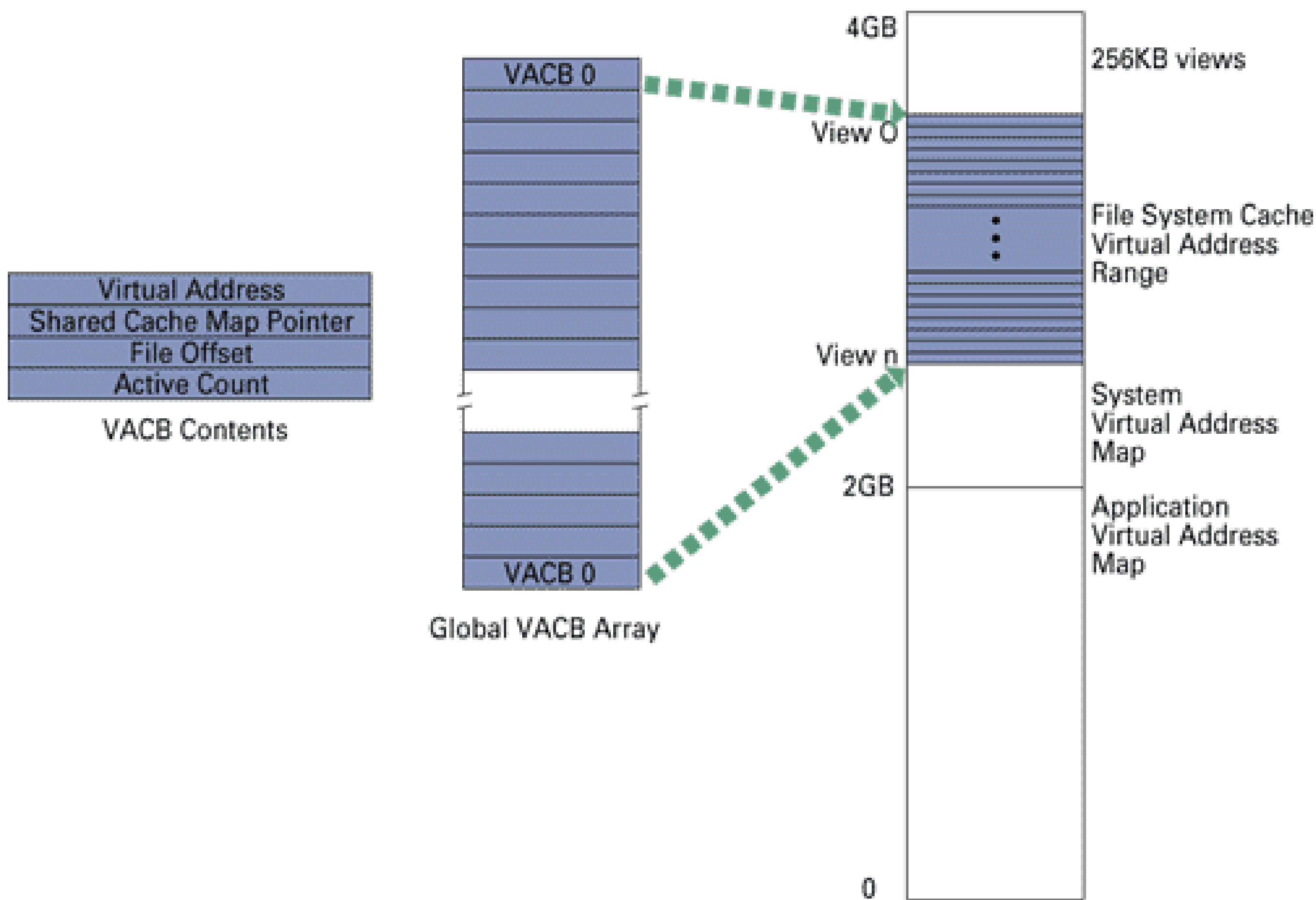


Virtual block caching caches data at the file system level rather than the disk level. When an application accesses data in a file, the file system checks to see whether the data resides in the cache. If the data is in the cache, the file system doesn't need to determine which sectors of the disk store the data and issue disk I/O requests. The file system simply operates on the data in the cache. NT relies on virtual block caching (implementing it in the Cache Manager), as do newer versions of UNIX, including Linux, Solaris, System V, and BSD 4.4. Virtual block caching has a couple of advantages over logical block caching. First, when file data the application is reading is in a virtual block cache, the file system performs no file-to-sector translations. In fact, in some cases, the I/O system can bypass the file system altogether and retrieve requested data directly from the cache. Second, the cache subsystem knows which files and which offsets within the files an application is asking for. The cache subsystem can monitor the access patterns of each file and make intelligent guesses about which data an application is going to ask for next. Using its guesses as guidelines, the cache subsystem reads the data from disk in anticipation of future requests. This slick process is known as read-ahead, and when the cache subsystem's predictions are accurate, read-ahead boosts system performance. Although read-ahead is possible with logical block caching, virtual block caching makes read-ahead simple to implement.

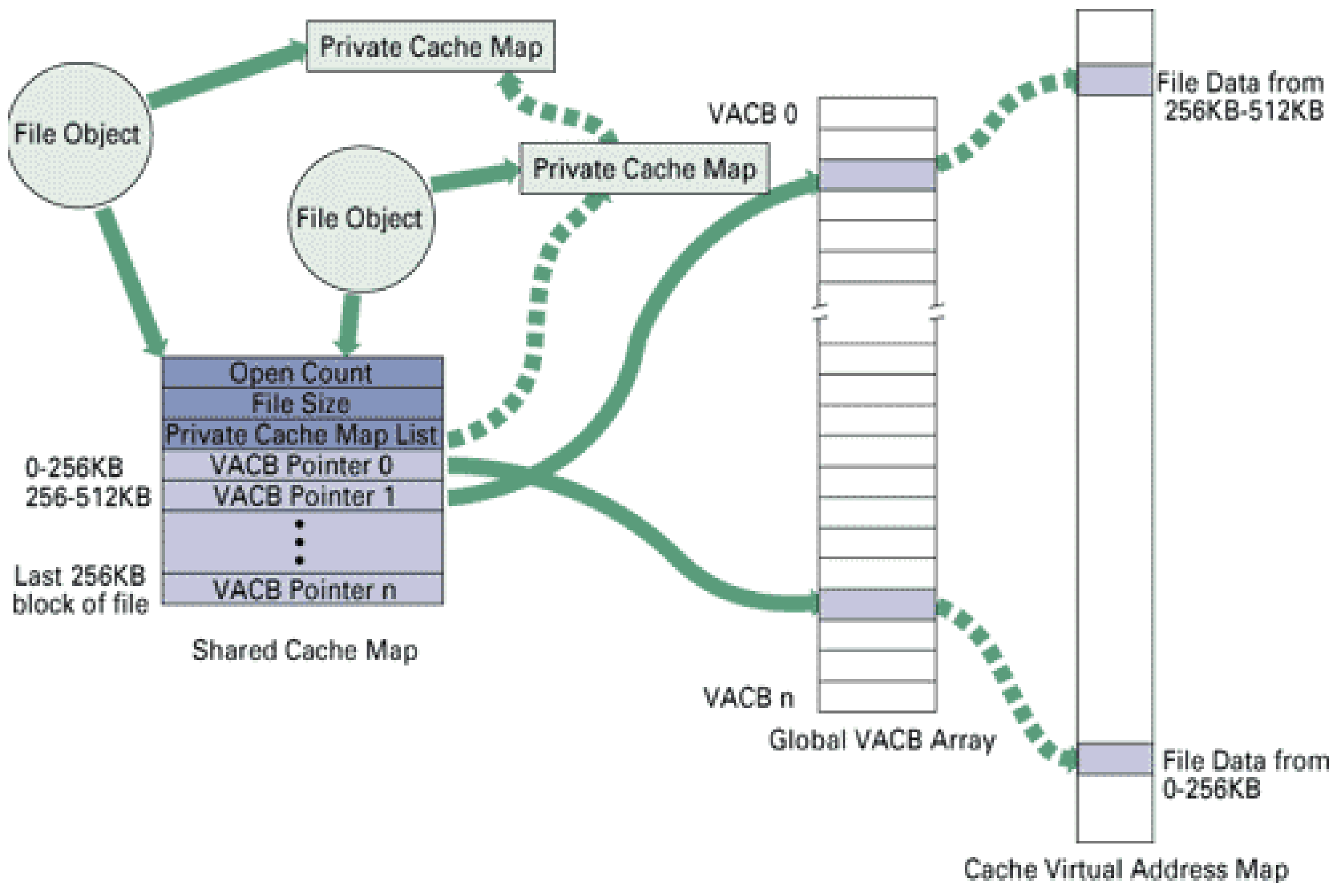
The Virtual Address Control Block Array

NT assigns the Cache Manager a fixed amount of a system's virtual address space (virtual memory is between 2GB and 4GB on most systems, and optionally between 3GB and 4GB on Enterprise Edition) during system initialization. Virtual memory is where the Cache Manager maps data in disk files, and the amount of virtual memory NT assigns to the cache depends on the size of physical memory. By default, NT gives the cache 128MB of virtual memory, but for each 4MB of physical memory above 16MB, NT gives the cache an additional 64MB of virtual memory. NT caps virtual memory at 512MB on x86 systems and 416MB on Alphas. (On x86 systems running NT 5.0 with Terminal Server support not enabled and with the Registry value HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/SessionManager/ MemoryManagement/LargeSystemCache set to 1, NT can assign the cache as much as 960MB of virtual memory.) Thus, if an x86 computer has 64MB of physical memory, NT sizes the cache at 512MB. The distinction between physical memory and virtual memory is important to remember, especially with respect to the cache: Although the Cache Manager might be tracking 512MB of virtual file-system data, the cache's working set is usually smaller. The file data present in the cache's working set--not the file data mapped into the cache's working memory--determines what data the Cache Manager caches.

After NT determines the cache's size, the Cache Manager initializes data structures to manage the cache's range of memory. The primary Cache Manager data structure is called the Virtual Address Control Block (VACB) array, which Figure 2 shows. NT logically divides the cache's virtual memory map into 256KB blocks and creates the VACB array with as many VACBs as there are 256KB blocks in the cache. For example, if the cache is 512MB in size, the VACB array will contain 2000 entries. Each entry in the VACB array records information about what 256KB portion of a file might map into the corresponding 256KB block of virtual memory in the cache. A VACB entry's contents include the address of the virtual memory block that corresponds to the entry. The other fields are valid only when part of a file maps into the virtual memory block. The File Offset is the 256KB-aligned offset into the file of the mapped data, and the Shared Cache Map Pointer references a Shared Cache Map structure NT uses to keep track on a per-file basis of which parts of a file are in the cache. The Active Count increments whenever an I/O is in progress on a file's data in that block of the cache.



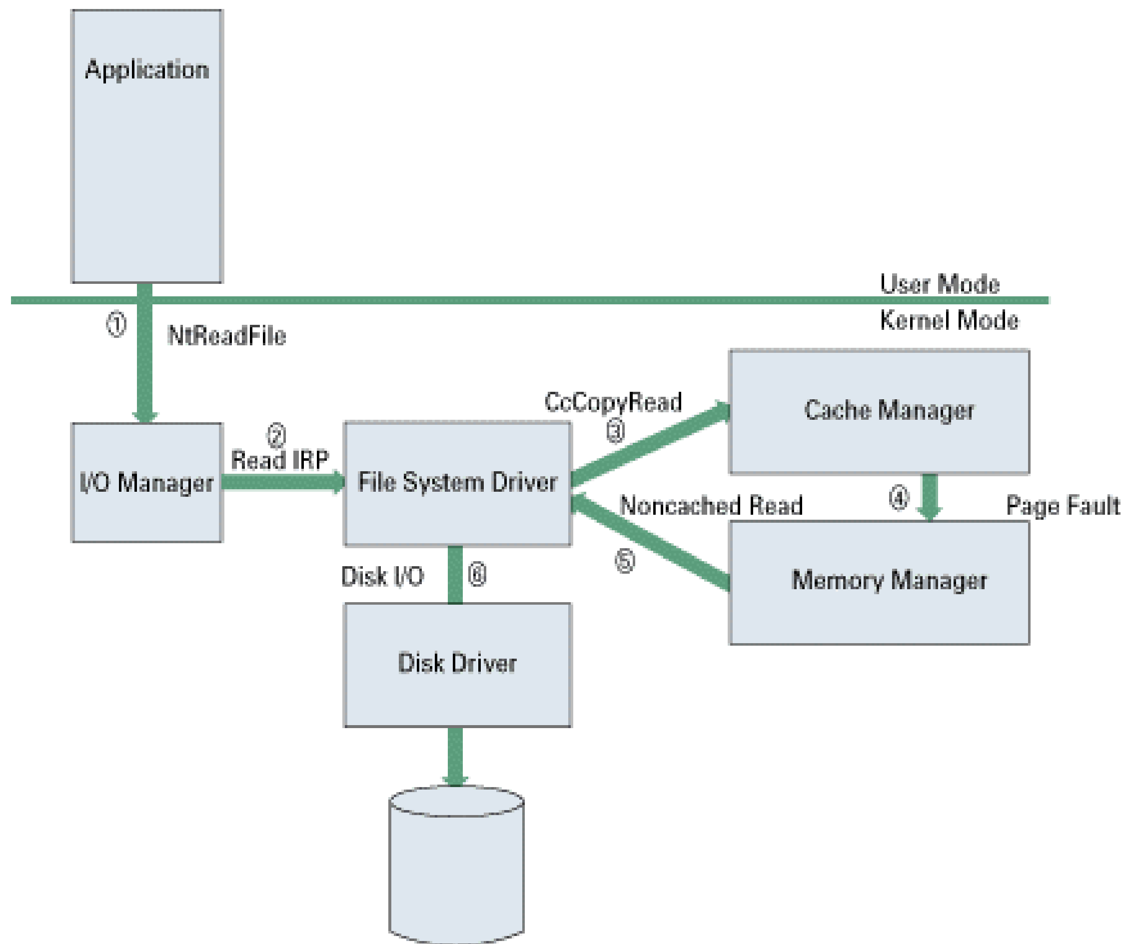
A file system initiates file caching when an application first reads to or writes from a file. By waiting for such a file operation before initiating file caching, the file system avoids unnecessary work if an application opens and closes a file without accessing the file's data. When the file system initiates file caching, it calls a Cache Manager function that checks to see whether an application has previously accessed the file. If the current access is the first, the Cache Manager allocates a Shared Cache Map data structure, such as you see in Figure 3. The reason NT calls this structure shared is because the Cache Manager allocates only one Shared Cache Map to each open file, and every subsequently created file object that represents an open instance of the file points at the file's unique Shared Cache Map. The Shared Cache Map helps the Cache Manager keep track of which parts of a file are mapped into the cache. The Shared Cache Map points to a list of other data structures (Private Cache Maps) and has an array of VACB pointers. A Private Cache Map stores hints for performing read-ahead. The VACB array in a file's Shared Cache Map is sized so that one VACB pointer represents each 256KB segment of the file. For example, a 345KB file has a VACB array consisting of two entries: one VACB pointer for the data between 0 and 256KB in the file, and a second VACB pointer for the data between 256KB and 345KB. If the Cache Manager mapped the file into the cache's virtual memory map, the VACB pointers in the Shared Cache Map's VACB pointer array point to the corresponding VACB entries in the Cache Manager's VACB array.



Cache Operation

The Cache Manager exports three interfaces that file systems use when they want to cache file data: the copy interface, the pin interface, and the Memory Descriptor List (MDL) interface. I'll describe each interface, but I will highlight the copy interface, which file systems use to service requests from applications running on the local machine.

Figure 4 illustrates the flow of control for a typical file-system request. The example path begins in an application that is reading 1000 bytes of data located at an offset of 300KB into a file that is 345KB in size. The I/O Manager takes the request and creates an I/O request packet (IRP) that it passes to the file system driver in charge of the disk containing the file. Because the application has not specified that the file shouldn't be cached, the file system driver initiates caching for the file. Then the Cache Manager creates a Shared Cache Map with two VACB pointers, both of which the Cache Manager initializes as not valid. The file system then calls the CcCopyRead function, which is part of the Cache Manager's copy interface. The file system driver can expect the Cache Manager to have retrieved the requested file data and copied it into the application's data buffer by the time the CcCopyRead function returns control to the file system driver.



Continuing with the example in Figure 4, the Cache Manager's CcCopyRead function must determine whether the file data is in the cache. First, CcCopyRead locates the file's Shared Cache Map through the Shared Cache Map pointer. The pointer is in the file object that the file system handed the CcCopyRead function to. Because an offset of 300KB lies in the second 256KB block of the file, the Cache Manager examines the second VACB pointer in the Shared Cache Map's VACB array. Finding this entry to be not valid, the Cache Manager scans through the global VACB array to find in the cache a 256KB block to which it can assign the file's data. The Cache Manager proceeds through the array, beginning with the entry just past the last one it assigned, looking for an entry that is not marked Active. When the Cache Manager finds such an entry, it marks the entry Active and checks to see whether any file has mapped data into the entry's portion of the cache. The Cache Manager examines the Shared Cache Map pointer in the VACB, and if the pointer is valid, the Cache Manager unmaps the cache slot's previous occupant. Next, the Cache Manager calls the Memory Manager to map a view of the file into the cache's virtual address map at the location the Cache Manager claimed, represented by the VACB pointer. In NT 5.0, VACBs will reside on a Least Recently Used list. The Cache Manager can go to the Least Recently Used list to locate VACBs, and it will try to use VACBs that have been inactive for the longest time. This change in NT 5.0 will provide a modest performance improvement.

After the Cache Manager calls the Memory Manager to map the file into the cache, the VACB's 256KB slot of virtual memory in the cache represents a 256KB block of data in the file (this segment in the file might be smaller than 256KB, so only part of the cache slot might be valid). Now all the Cache Manager must do is perform a simple memory copy from the cache memory to the application's buffer.

The first time the application accesses each page of the file's data, the access generates a page fault because no physical memory backs the virtual memory. Each page fault invokes the Memory Manager, which determines that the virtual memory corresponds to a portion of the file and allocates physical memory to back the virtual memory. Then, the Memory Manager calls back the file system driver that called the Cache Manager, and the Memory Manager instructs the file system driver to read the file's contents from disk into virtual memory (and therefore physical memory). After the file system driver reads the file's contents into virtual memory, the Memory Manager continues to execute the Cache Manager, which successfully copies the data into the application's buffer. After the copy completes, the Active count in the VACB decrements.

An interesting aspect of the copy interface is that, as the example above shows, NT can invoke a file system driver to fetch data that the same file system driver requested from the Cache Manager. The driver does not call the Cache Manager when the Memory Manager invokes the driver, because the I/O requests that the Memory Manager's page fault routine gives the file system driver are marked noncached request. Any time a file system driver receives noncached requests, the driver will not look for the data in the cache but instead will perform disk operations.

The path for a write operation is almost identical to the read operation path I've just described. Instead of calling CcCopyRead, the file system calls CcCopyWrite, and instead of copying from the cache to the application's buffer, the Cache Manager copies from the application's buffer to the cache. The Cache Manager notes in the Shared Cache Map that the data is dirty and must eventually be written, with the aid of a file system driver, back to the file on disk.

In the read operation example in Figure 4, an application was reading the file for the first time, so the file's data was not stored in the cache. If the same or another application subsequently accesses the same portion of the file, the VACB pointer in the file's Shared Cache Map will be valid. In that case, the Cache Manager does not need to find a free VACB to assign to the file. The Cache Manager can simply use the virtual memory already assigned and perform the copy. However, using virtual memory does not mean that the copy operation won't incur page faults, because the Memory Manager might have removed from the cache's working set the physical memory reserved for the virtual memory, or it might have moved the physical memory to another part of the cache.

File systems use the Cache Manager's pin interface to cache their file system metadata. Metadata includes the FAT in the FAT file system and the Master File Table (MFT) in the NTFS file system. A file system tells the Cache Manager that it is caching metadata, and the file system can reference virtual addresses within the cache to access the metadata. This procedure lets file systems modify their metadata directly in the cache, rather than copy metadata back and forth to private buffers, as they would need to do if they used the copy interface. To ensure that the virtual memory in the cache representing the metadata remains valid for the duration of the file system's access, the file system directs the Cache Manager to mark the VACBs representing the metadata as active. To do so, the file system calls the CcPinRead, CcPinMappedData, or CcPinPrepareWrite function. The Cache Manager uses a special Memory Manager function, MmCheckPageCachedState, to lock memory. When the file system is finished accessing the metadata, the file system calls CcUnpinData to notify the Cache Manager to decrement the VACB Active counts. One caveat to this interface is that, for most metadata files, the file system takes over the role of telling the Cache Manager when it needs to write any dirty data back to disk. Modified data therefore remains in physical memory until the file system tells the Cache Manager to flush the data to disk.

Network file server file systems such as NT's built-in file server, SRV, use the MDL interface. In response to a network request to read from or write to a file, SRV invokes either the CcMdlRead or

CcMdlPrepareWrite Cache Manager functions. The Cache Manager assigns the requested data a slot in the cache and locks the virtual memory into physical memory. Rather than return a virtual address to the file system, as the pin interface does, the MDL interface returns a description of the physical memory that the Memory Manager assigns the cached data. The MDL interface formats the physical memory description as an MDL. Network protocol drivers (such as the TCP/IP driver) and network card interface drivers require data that is locked into physical memory and MDLs that describe the data, so the Cache Manager can hand the MDLs directly to the networking software. No data copying is necessary to send file system data across the network, because the Cache Manager can send the data from the cache. When the file system is finished accessing the data, the file system calls CcMdlReadComplete or CcMdlWriteComplete to unlock the data.

The Fast I/O Interface

NT's use of virtual block caching means that, under certain circumstances, the I/O Manager need not create an IRP when servicing an application's file request. A file system driver can set a flag in the file's file object to alert the I/O Manager that Fast I/O (Microsoft originally called this interface Turbo I/O) is possible for a file. When the I/O Manager sees this set flag, it will call the file system driver directly instead of building an IRP to send to the driver.

During its initialization, a file system driver must register its Fast I/O functions with the I/O Manager. Most file system drivers accomplish this registration by pointing their Fast I/O functions to corresponding functions in the Cache Manager. The I/O Manager checks the return status after it calls one of the file system driver's Fast I/O functions to read, write, or perform other file system requests. If the I/O Manager receives a FALSE in response to its call, it will build an IRP. A FALSE response generally means that the requested data is not entirely in the cache and that disk I/O is necessary to fulfill the request. Thus, the I/O Manager can avoid the overhead of building an IRP if the file data is entirely in the cache.

File system drivers usually disable the Fast I/O interface for a particular file when the drivers must perform processing in addition to reading or writing data to service a request. One situation in which Fast I/O is not possible on a file is when an application locks a portion of a file for exclusive access. The file system driver must honor such locks by examining carefully the ranges of all requests to the file and blocking those ranges that overlap the locked areas.

Read-Ahead and Lazy Writing

The Cache Manager keeps information about the two most recent reads on a file in the file object's Private Cache Map data structure. When an application reads from the file, the Cache Manager compares the current request with the previous two in the Private Cache Map. If the Cache Manager detects a pattern in the requests, it will extrapolate the pattern to read not only the data that the application is requesting, but also the data at the next location in the pattern. The Cache Manager fetches the application's requested data immediately but also initiates an extrapolated read in the background (asynchronously), which a thread in the system's pool of worker threads performs. When this read-ahead process is effective, an application reading a file will always find the file data in the cache, because the data is preread. This technique works when an application reads a file's data sequentially and also when the application skips around the file in a regular manner. Also, an application can inform the Cache Manager when the application opens a file that it will read the file sequentially.

If the Cache Manager makes no effort to write modified file data back to a file, the Memory Manager's modified writer thread will write the unwritten data back if free memory becomes scarce. Generally, the system doesn't rely on the Memory Manager to flush file data back to the disk. Instead, the Cache

Manager tries to get the data back to nonvolatile storage in a timely manner using a process called lazy writing. As applications modify file data, the Cache Manager keeps track of how much data is dirty. Once every second, the Cache Manager writes back one-quarter of the cache's dirty data to disk. To accumulate data to write, the Cache Manager traverses the Shared Cache Map linked data structures that reference VACBs corresponding to dirty regions in the cache, and scans the VACB pointer arrays. If a VACB pointer points to a VACB marked as dirty, the Cache Manager calls a Memory Manager function, MmFlushSection, to initiate a Memory Manager write-back of the associated cache virtual memory. The Memory Manager then calls the file system responsible for the data to write the modified portions of the file to disk and to note that the pages are clean. At that time, the Cache Manager marks as clean the VACB for which data was written.

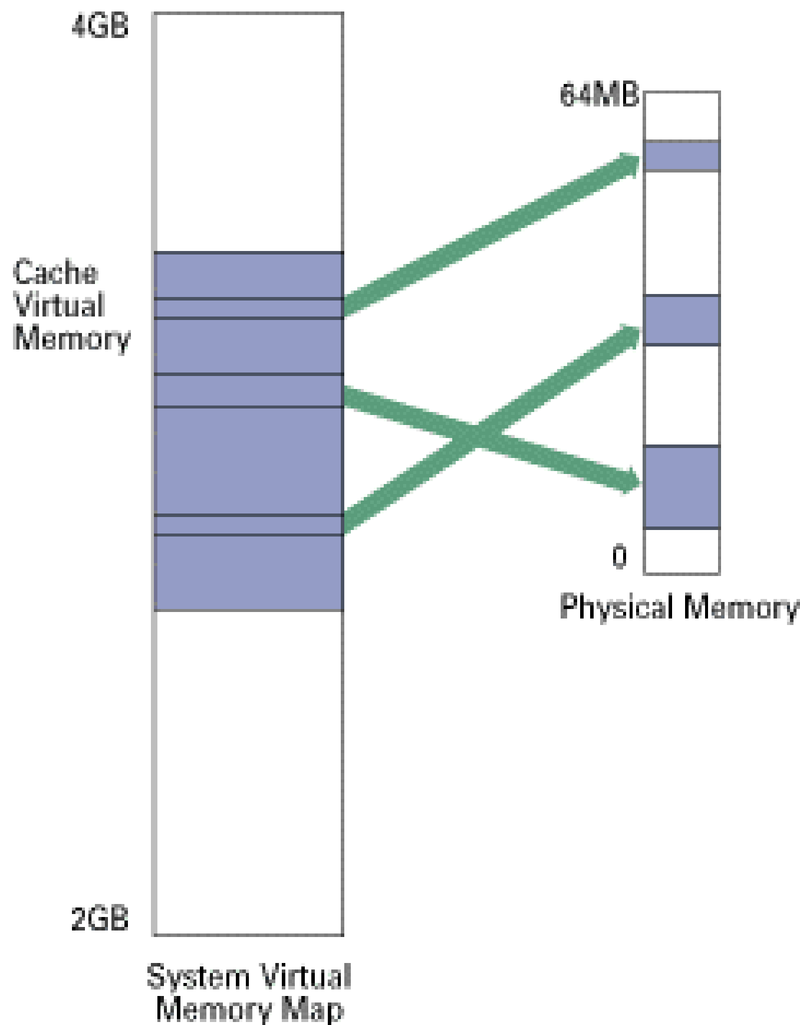
If too much modified data accumulates, the overall performance of a system can degrade. To prevent this situation, file system drivers notify the Cache Manager when they are ready to write file system data. The file system driver calls the Cache Manager's CcCanIWrite function and specifies how much data the driver will write. The CcCanIWrite function looks at the amount of dirty data accumulated in the cache and at the amount of data the driver will write. If the total amount of this data is greater than a threshold the Cache Manager keeps, the Cache Manager will immediately initiate a flush of dirty file data back to disk. The Cache Manager bases the threshold on how much physical memory the system contains. When the modified data in the cache is below the threshold, the Cache Manager lets the file system driver proceed with its write operation. NT calls this process of regulating the rate at which dirty cache data accumulates write-throttling.

Controlling File Caching

NT gives applications a great deal of control over how and when the Cache Manager caches a file's data. When an application opens a file, the application can request that the Cache Manager not cache the file at all, or that the Cache Manager immediately reflect on disk any modifications to the file. In addition, an application can direct the Cache Manager at any time to flush a file's cached contents back to disk. Finally, applications can open temporary files. For example, an application setup program will uncompress its setup package into temporary files. The Cache Manager will not include any modified data from temporary files in its lazy write operations, and applications delete temporary files after the data in the files have served their purpose.

The Cache's Working Set

I stated earlier that the cache consists of the file data present in physical memory, not the data mapped in the Cache Manager's virtual address map. Just as typical applications have, the Cache Manager has a working set that the Memory Manager increases and decreases as the memory demands of active system processes dictate. NT shares the cache's working set with the memory assigned to pageable kernel and device driver code and data, in addition to the system's paged pool. Various Executive subsystems use paged pool memory to allocate data structures. Figure 5 demonstrates how the cache's working set is usually smaller than the cache's virtual memory map.



The Memory Manager determines the maximum size of the cache's working set (and of the paged pool) during system initialization. If you set the Registry key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\MemoryManagement\ LargeSystemCache to 1 on pre-Service Pack 4 (SP4) NT 4.0 systems, the cache's working set can grow (if the memory demands of other processes are small) to the size of postboot available physical memory minus 4MB. Postboot available physical memory is the size of physical memory minus the memory NT assigns to the kernel image, code, and boot-time device drivers during the boot. In NT 4.0 SP4 and NT 5.0, a LargeSystemCache value of 1 lets the cache grow only to the postboot available physical memory minus 32MB, and then only if an x86 system has more than 128MB of memory (256MB of memory on an Alpha system). Otherwise, the cache's working set maximum size is smaller and based on the amount of physical memory on the system. For example, a machine with 64MB of memory will have a maximum cache working set size of 8MB.

The Memory Manager will increase the working set beyond the maximum if ample free memory exists. The working set assigned to the cache can never grow larger than the virtual size of the cache. If the cache size during system initialization is 512MB, then the Memory Manager will assign a maximum of 512MB of physical memory to the cache, even if the system has 4GB of memory and much of it is free. However, the effective size of the physical memory that stores file data can be greater than the cache's virtual size. As I mentioned in last month's column, the Memory Manager removes pages

from a working set and places them on the standby list--their contents remain in physical memory as long as another process does not allocate memory and reassign the pages. If the Memory Manager removes pages from the Cache Manager's working set and memory is plentiful, file data will reside in physical memory. If the Cache Manager reassigns virtual memory to represent the same portion of the file that has physical pages on the standby list, the Memory Manager quickly assigns memory back to the Cache Manager without requiring any disk I/O.

The Memory Manager maintains a close tie with the Cache Manager through working-set-trimming code. When the Memory Manager is scanning the pages in the working set of the Cache Manager to find pages to remove, the Memory Manager will ignore pages that the Cache Manager's pin and MDL interfaces have locked into memory.

NT's file system cache provides an efficient means to keep frequently accessed portions of files resident in physical memory for fast reads and writes. The tight association between file system drivers, the Cache Manager, and the Memory Manager enables intelligent read-ahead and a fast I/O path and lets the cache's physical memory usage grow and shrink according to the same algorithms that manage application working sets.