11.5.3 Implementation of Memory Management

Windows, on the x86, supports a single linear 4-GB demand-paged address space per process. Segmentation is not supported in any form. Theoretically, page sizes can be any power of 2 up to 64 KB. On the x86 they are normally fixed at 4 KB. In addition, the operating system can use 2-MB large pages to improve the effectiveness of the **TLB** (**Translation Lookaside Buffer**) in the processor's memory management unit. Use of 2-MB large pages by the kernel and large applications significantly improves performance by improving the hit rate for the TLB and reducing the number of times the page tables have to be walked to find entries that are missing from the TLB.

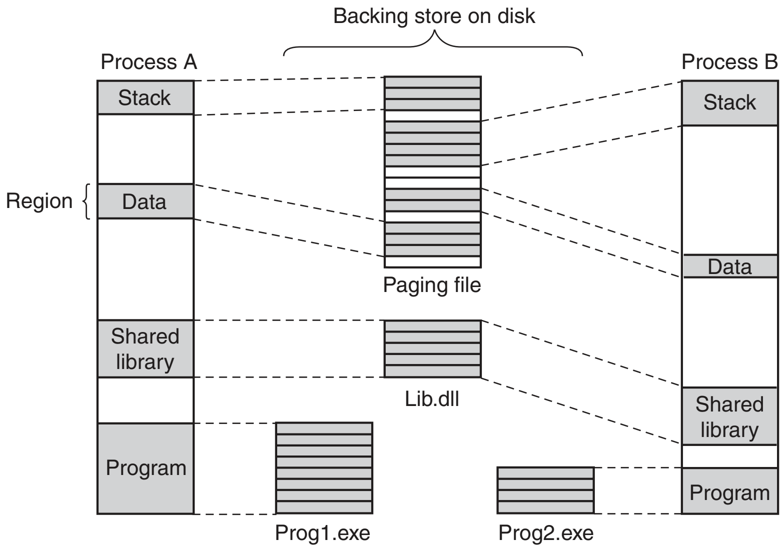


Figure 11-30. Mapped regions with their shadow pages on disk. The *lib.dll* file is mapped into two address spaces at the same time.

Unlike the scheduler, which selects individual threads to run and does not care much about processes, the memory manager deals entirely with processes and does not care much about threads. After all, processes, not threads, own the address space and that is what the memory manager is concerned with. When a region of virtual address space is allocated, as four of them have been for process *A* in Fig. 11-30, the memory manager creates a **VAD** (**Virtual Address Descriptor**) for it, listing the range of addresses mapped, the section representing the backing store file and offset where it is mapped, and the permissions. When the first page is touched, the directory of page tables is created and its physical address is inserted into the process object. An address space is completely defined by the list of its VADs. The VADs are organized into a balanced tree, so that the descriptor for a particular address can be found efficiently. This scheme supports sparse address spaces. Unused areas between the mapped regions use no resources (memory or disk) so they are essential free.

Page-Fault Handling

When a process starts on Windows, many of the pages mapping the program's EXE and DLL image files may already be in memory because they are shared with other processes. The writable pages of the images are marked *copy-on-write* so that they can be shared up to the point they need to be modified. If the operating system recognizes the EXE from a previous execution, it may have recorded the page-reference pattern, using a technology Microsoft calls **SuperFetch**. SuperFetch attempts to prepage many of the needed pages even though the process has not faulted on them yet. This reduces the latency for starting up applications by overlapping the reading of the pages from disk with the execution of the initialization code in the images. It improves throughput to disk because it is easier for the disk drivers to organize the reads to reduce the seek time needed. Process prepaging is also used during boot of the system, when a background application moves to the foreground, and when restarting the system after hibernation.

Prepaging is supported by the memory manager, but implemented as a separate component of the system. The pages brought in are not inserted into the process' page table, but instead are inserted into the *standby list* from which they can quickly be inserted into the process as needed without accessing the disk.

Nonmapped pages are slightly different in that they are not initialized by reading from the file. Instead, the first time a nonmapped page is accessed the memory manager provides a new physical page, making sure the contents are all zeroes (for security reasons). On subsequent faults a nonmapped page may need to be found in memory or else must be read back from the pagefile.

Demand paging in the memory manager is driven by page faults. On each page fault, a trap to the kernel occurs. The kernel then builds a machine-independent descriptor telling what happened and passes this to the memory-manager part of the executive. The memory manager then checks the access for validity. If the faulted page falls within a committed region, it looks up the address in the list of VADs and finds (or creates) the process page-table entry. In the case of a shared page, the memory manager uses the prototype page-table entry associated with the section object to fill in the new page-table entry for the process page table.

The format of the page-table entries differs depending on the processor architecture. For the x86 and x64, the entries for a mapped page are shown in Fig. 11-31. If an entry is marked valid, its contents are interpreted by the hardware so that the virtual address can be translated into the correct physical page. Unmapped pages also have entries, but they are marked *invalid* and the hardware ignores the rest of the entry. The software format is somewhat different from the hardware format and is determined by the memory manager. For example, for an unmapped page that must be allocated and zeroed before it may be used, that fact is noted in the page-table entry.

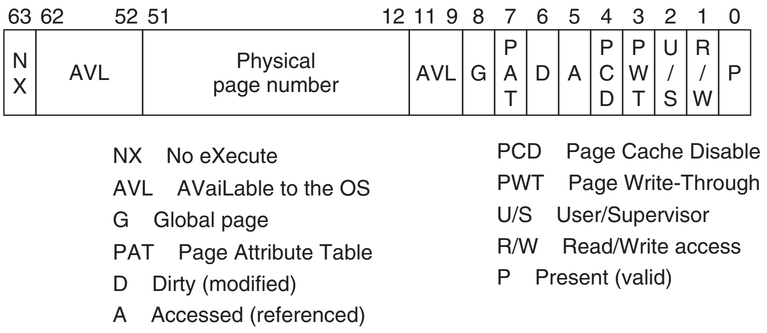


Figure 11-31. A page-table entry (PTE) for a mapped page on the Intel x86 and AMD x64 architectures.

Two important bits in the page-table entry are updated by the hardware directly. These are the access (A) and dirty (D) bits. These bits keep track of when a particular page mapping has been used to access the page and whether that access could have modified the page by writing it. This really helps the performance of the system because the memory manager can use the access bit to implement the **LRU** (**Least-Recently Used**) style of paging. The LRU principle says that pages which have not been used the longest are the least likely to be used again soon. The access bit allows the memory manager to determine that a page has been accessed. The dirty bit lets the memory manager know that a page may have been modified, or more significantly, that a page has *not* been modified. If a page has not been modified since being read from disk, the memory manager does not have to write the contents of the page to disk before using it for something else.

Both the x86 and x64 use a 64-bit page-table entry, as shown in Fig. 11-31.

Each page fault can be considered as being in one of five categories:

1. The page referenced is not committed.
2. Access to a page has been attempted in violation of the permissions.
3. A shared copy-on-write page was about to be modified.
4. The stack needs to grow.
5. The page referenced is committed but not currently mapped in.

The first and second cases are due to programming errors. If a program attempts to use an address which is not supposed to have a valid mapping, or attempts an invalid operation (like attempting to write a read-only page) this is called an **access violation** and usually results in termination of the process. Access violations are often the result of bad pointers, including accessing memory that was freed and unmapped from the process.

The third case has the same symptoms as the second one (an attempt to write to a read-only page), but the treatment is different. Because the page has been marked as *copy-on-write*, the memory manager does not report an access violation, but instead makes a private copy of the page for the current process and then returns control to the thread that attempted to write the page. The thread will retry the write, which will now complete without causing a fault.

The fourth case occurs when a thread pushes a value onto its stack and crosses onto a page which has not been allocated yet. The memory manager is programmed to recognize this as a special case. As long as there is still room in the virtual pages reserved for the stack, the memory manager will supply a new physical page, zero it, and map it into the process. When the thread resumes running, it will retry the access and succeed this time around.

Finally, the fifth case is a normal page fault. However, it has several subcases. If the page is mapped by a file, the memory manager must search its data structures, such as the prototype page table associated with the section object to be sure that there is not already a copy in memory. If there is, say in another process or on the standby or modified page lists, it will just share it—perhaps marking it as copy-on-write if changes are not supposed to be shared. If there is not already a copy, the memory manager will allocate a free physical page and arrange for the file page to be copied in from disk, unless another the page is already transitioning in from disk, in which case it is only necessary to wait for the transition to complete.

When the memory manager can satisfy a page fault by finding the needed page in memory rather than reading it in from disk, the fault is classified as a **soft fault**. If the copy from disk is needed, it is a **hard fault**. Soft faults are much cheaper, and have little impact on application performance compared to hard faults. Soft faults can occur because a shared page has already been mapped into another process, or only a new zero page is needed, or the needed page was trimmed from the process' working set but is being requested again before it has had a chance to be reused. Soft faults can also occur because pages have been compressed to effectively increase the size of physical memory. For most configurations of CPU, memory, and I/O in current systems it is more efficient to use compression rather than incur the I/O expense (performance and energy) required to read a page from disk.

When a physical page is no longer mapped by the page table in any process it goes onto one of three lists: free, modified, or standby. Pages that will never be needed again, such as stack pages of a terminating process, are freed immediately. Pages that may be faulted again go to either the modified list or the standby list, depending on whether or not the dirty bit was set for any of the page-table entries that mapped the page since it was last read from disk. Pages in the modified list will be eventually written to disk, then moved to the standby list.

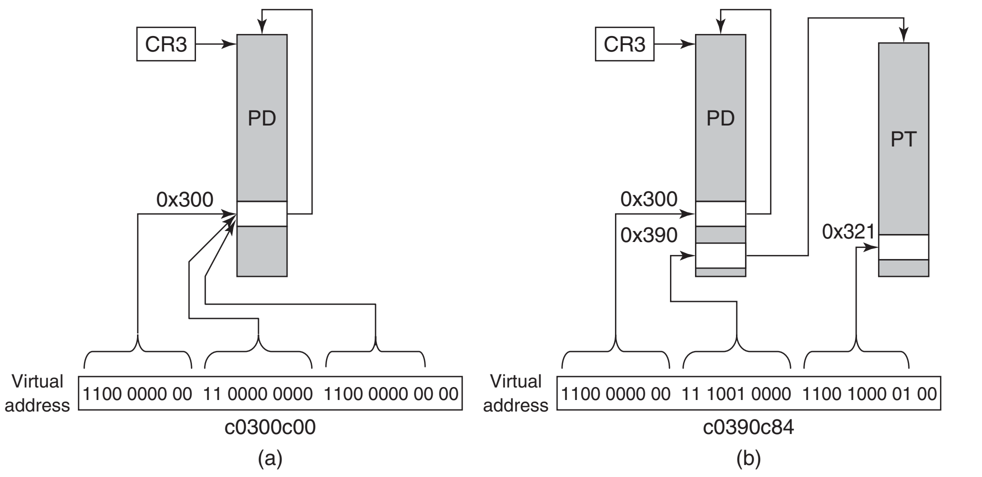
The memory manager can allocate pages as needed using either the free list or the standby list. Before allocating a page and copying it in from disk, the memory manager always checks the standby and modified lists to see if it already has the page in memory. The prepaging scheme in Windows thus converts future hard faults into soft faults by reading in the pages that are expected to be needed and pushing them onto the standby list. The memory manager itself does a small amount of ordinary prepaging by accessing groups of consecutive pages rather than single pages. The additional pages are immediately put on the standby list. This is not generally wasteful because the overhead in the memory manager is very much dominated by the cost of doing a single I/O. Reading a cluster of pages rather than a single page is negligibly more expensive.

The page-table entries in Fig. 11-31 refer to physical page numbers, not virtual page numbers. To update page-table (and page-directory) entries, the kernel needs to use virtual addresses. Windows maps the page tables and page directories for the current process into kernel virtual address space using self-map entries in the page directory, as shown in Fig. 11-32. By making page-directory entries point at the page directory (the self-map), there are virtual addresses that can be used to refer to page-directory entries (a) as well as page table entries (b). The self-map occupies the same 8 MB of kernel virtual addresses for every process (on the x86). For simplicity the figure shows the x86 self-map for 32-bit **PTEs** (**Page-Table Entries**). Windows actually uses 64-bit PTEs so the system can makes use of more than 4 GB of physical memory. With 32-bit PTEs, the self-map uses only one **PDE** (**Page-Directory Entry**) in the page directory, and thus occupies only 4 MB of addresses rather than 8 MB.

The Page Replacement Algorithm

When the number of free physical memory pages starts to get low, the memory manager starts working to make more physical pages available by removing them from user-mode processes as well as the system process, which represents kernel-mode use of pages. The goal is to have the most important virtual pages present in memory and the others on disk. The trick is in determining what *important* means. In Windows this is answered by making heavy use of the working-set concept. Each process (*not* each thread) has a working set. This set consists of the mapped-in pages that are in memory and thus can be referenced without a page fault. The size and composition of the working set fluctuates as the process' threads run, of course.

Each process' working set is described by two parameters: the minimum size and the maximum size. These are not hard bounds, so a process may have fewer pages in memory than its minimum or (under certain circumstances) more than its maximum. Every process starts with the same minimum and maximum, but these bounds can change over time, or can be determined by the job object for processes contained in a job. The default initial minimum is in the range 20–50 pages and



Self-map: PD[0xc0300000>>22] is PD (page-directory)
Virtual address (a): (PTE *)(0xc0300c00) points to PD[0x300] which is the self-map page directory entry
Virtual address (b): (PTE *)(0xc0390c84) points to PTE for virtual address 0xe4321000

Figure 11-32. The Windows self-map entries are used to map the physical pages of the page tables and page directory into kernel virtual addresses (shown for 32-bit PTEs).

the default initial maximum is in the range 45–345 pages, depending on the total amount of physical memory in the system. The system administrator can change these defaults, however. While few home users will try, server admins might.

Working sets come into play only when the available physical memory is getting low in the system. Otherwise processes are allowed to consume memory as they choose, often far exceeding the working-set maximum. But when the system comes under **memory pressure**, the memory manager starts to squeeze processes back into their working sets, starting with processes that are over their maximum by the most. There are three levels of activity by the working-set manager, all of which is periodic based on a timer. New activity is added at each level:

1. **Lots of memory available:** Scan pages resetting access bits and using their values to represent the *age* of each page. Keep an estimate of the unused pages in each working set.

2. **Memory getting tight:** For any process with a significant proportion of unused pages, stop adding pages to the working set and start replacing the oldest pages whenever a new page is needed. The replaced pages go to the standby or modified list.

3. **Memory is tight:** Trim (i.e., reduce) working sets to be below their maximum by removing the oldest pages.

The working set manager runs every second, called from the **balance set manager** thread. The working-set manager throttles the amount of work it does to keep from overloading the system. It also monitors the writing of pages on the modified list to disk to be sure that the list does not grow too large, waking the ModifiedPageWriter thread as needed.

Physical Memory Management

Above we mentioned three different lists of physical pages, the free list, the standby list, and the modified list. There is a fourth list which contains free pages that have been zeroed. The system frequently needs pages that contain all zeros. When new pages are given to processes, or the final partial page at the end of a file is read, a zero page is needed. It is time consuming to write a page with zeros, so it is better to create zero pages in the background using a low-priority thread. There is also a fifth list used to hold pages that have been detected as having hardware errors (i.e., through hardware error detection).

All pages in the system either are referenced by a valid page-table entry or are on one of these five lists, which are collectively called the **PFN database** (**Page Frame Number database**). Fig. 11-33 shows the structure of the PFN Database. The table is indexed by physical page-frame number. The entries are fixed length, but different formats are used for different kinds of entries (e.g., shared vs. private). Valid entries maintain the page's state and a count of how many page tables point to the page, so that the system can tell when the page is no longer in use. Pages that are in a working set tell which entry references them. There is also a pointer to the process page table that points to the page (for nonshared pages) or to the prototype page table (for shared pages).

Additionally there is a link to the next page on the list (if any), and various other fields and flags, such as *read in progress*, *write in progress*, and so on. To save space, the lists are linked together with fields referring to the next element by its index within the table rather than pointers. The table entries for the physical pages are also used to summarize the dirty bits found in the various page table entries that point to the physical page (i.e., because of shared pages). There is also information used to represent differences in memory pages on larger server systems which have memory that is faster from some processors than from others, namely NUMA machines.

Pages are moved between the working sets and the various lists by the working-set manager and other system threads. Let us examine the transitions. When the working-set manager removes a page from a working set, the page goes on the bottom of the standby or modified list, depending on its state of cleanliness. This transition is shown as (1) in Fig. 11-34.

Pages on both lists are still valid pages, so if a page fault occurs and one of these pages is needed, it is removed from the list and faulted back into the working set without any disk I/O (2). When a process exits, its nonshared pages cannot be

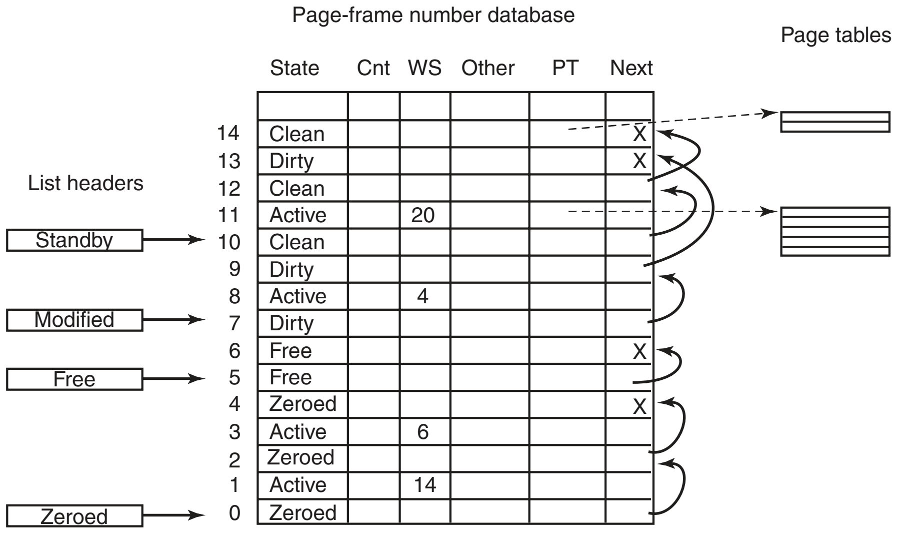


Figure 11-33. Some of the major fields in the page-frame database for a valid page.

faulted back to it, so the valid pages in its page table and any of its pages on the modified or standby lists go on the free list (3). Any pagefile space in use by the process is also freed.

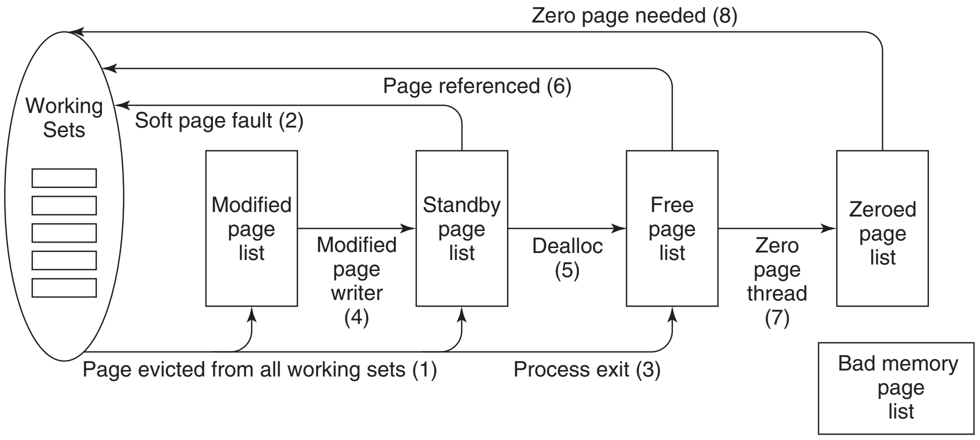


Figure 11-34. The various page lists and the transitions between them.

Other transitions are caused by other system threads. Every 4 seconds the balance set manager thread runs and looks for processes all of whose threads have been idle for a certain number of seconds. If it finds any such processes, their kernel stacks are unpinned from physical memory and their pages are moved to the standby or modified lists, also shown as (1).

Two other system threads, the **mapped page writer** and the **modified page writer**, wake up periodically to see if there are enough clean pages. If not, they take pages from the top of the modified list, write them back to disk, and then move them to the standby list (4). The former handles writes to mapped files and the latter handles writes to the pagefiles. The result of these writes is to transform modified (dirty) pages into standby (clean) pages.

The reason for having two threads is that a mapped file might have to grow as a result of the write, and growing it requires access to on-disk data structures to allocate a free disk block. If there is no room in memory to bring them in when a page has to be written, a deadlock could result. The other thread can solve the problem by writing out pages to a paging file.

The other transitions in Fig. 11-34 are as follows. If a process unmaps a page, the page is no longer associated with a process and can go on the free list (5), except for the case that it is shared. When a page fault requires a page frame to hold the page about to be read in, the page frame is taken from the free list (6), if possible. It does not matter that the page may still contain confidential information because it is about to be overwritten in its entirety.

The situation is different when a stack grows. In that case, an empty page frame is needed and the security rules require the page to contain all zeros. For this reason, another kernel system thread, the **ZeroPage thread**, runs at the lowest priority (see Fig. 11-26), erasing pages that are on the free list and putting them on the zeroed page list (7). Whenever the CPU is idle and there are free pages, they might as well be zeroed since a zeroed page is potentially more useful than a free page and it costs nothing to zero the page when the CPU is idle.

The existence of all these lists leads to some subtle policy choices. For example, suppose that a page has to be brought in from disk and the free list is empty. The system is now forced to choose between taking a clean page from the standby list (which might otherwise have been faulted back in later) or an empty page from the zeroed page list (throwing away the work done in zeroing it). Which is better?

The memory manager has to decide how aggressively the system threads should move pages from the modified list to the standby list. Having clean pages around is better than having dirty pages around (since clean ones can be reused instantly), but an aggressive cleaning policy means more disk I/O and there is some chance that a newly cleaned page may be faulted back into a working set and dirtied again anyway. In general, Windows resolves these kinds of trade-offs through algorithms, heuristics, guesswork, historical precedent, rules of thumb, and administrator-controlled parameter settings.

Modern Windows introduced an additional abstraction layer at the bottom of the memory manager, called the **store manager**. This layer makes decisions about how to optimize the I/O operations to the available backing stores. Persistent storage systems include auxiliary flash memory and SSDs in addition to rotating disks.

The store manager optimizes where and how physical memory pages are backed by the persistent stores in the system. It also implements optimization techniques such as copy-on-write sharing of identical physical pages and compression of the pages in the standby list to effectively increase the available RAM.

Another change in memory management in Modern Windows is the introduction of a **swap file**. Historically memory management in Windows has been based on working sets, as described above. As memory pressure increases, the memory manager squeezes on the working sets to reduce the footprint each process has in memory. The modern application model introduces opportunities for new efficiencies. Since the process containing the foreground part of a modern application is no longer given processor resources once the user has switched away, there is no need for its pages to be resident. As memory pressure builds in the system, the pages in the process may be removed as part of normal working-set management. However, the process lifetime manager knows how long it has been since the user switched to the application's foreground process. When more memory is needed it picks a process that has not run in a while and calls into the memory manager to efficiently swap all the pages in a small number of I/O operations. The pages will be written to the swap file by aggregating them into one or more large chunks. This means that the entire process can also be restored in memory with fewer I/O operations.

All in all, memory management is a highly complex executive component with many data structures, algorithms, and heuristics. It attempts to be largely self tuning, but there are also many knobs that administrators can tweak to affect system performance. A number of these knobs and the associated counters can be viewed using tools in the various tool kits mentioned earlier. Probably the most important thing to remember here is that memory management in real systems is a lot more than just one simple paging algorithm like clock or aging.

11.6 CACHING IN WINDOWS

The Windows cache improves the performance of file systems by keeping recently and frequently used regions of files in memory. Rather than cache physical addressed blocks from the disk, the cache manager manages virtually addressed blocks, that is, regions of files. This approach fits well with the structure of the native NT File System (NTFS), as we will see in Sec. 11.8. NTFS stores all of its data as files, including the file-system metadata.

The cached regions of files are called *views* because they represent regions of kernel virtual addresses that are mapped onto file-system files. Thus, the actual management of the physical memory in the cache is provided by the memory manager. The role of the cache manager is to manage the use of kernel virtual addresses for views, arrange with the memory manager to *pin* pages in physical memory, and provide interfaces for the file systems.