Memory Management

- Windows 3.x vs Win32
- Dynamic memory allocation
- Shared memory
- File mapping
- Memory heaps
- Global and local heap memory
- C run-time memory

The 32-bit Windows NT operating system offers a similar Windows programming interface to the 16-bit Windows 3.x; there is considerable source code compatibility between the two. However, Windows 3.x is heavily tied to the Intel family of 80*86 processors, whereas Windows NT is very much a multi-platform operating system, designed to port easily to any 32-bit processor. It already runs on Intel's 80386/80486, MIPS R4000 and DEC Alpha AXP processors. So you would expect the area of memory management to be quite different between Windows 3.x and Windows NT and you would be right!

Windows NT is designed to offer 32-bit demand-paged virtual memory, and all the processors mentioned above support this feature. In *'flat memory model'*, each application has its own private 4 GB logical address space and memory allocation is performed on a uniform page basis rather than using segments.

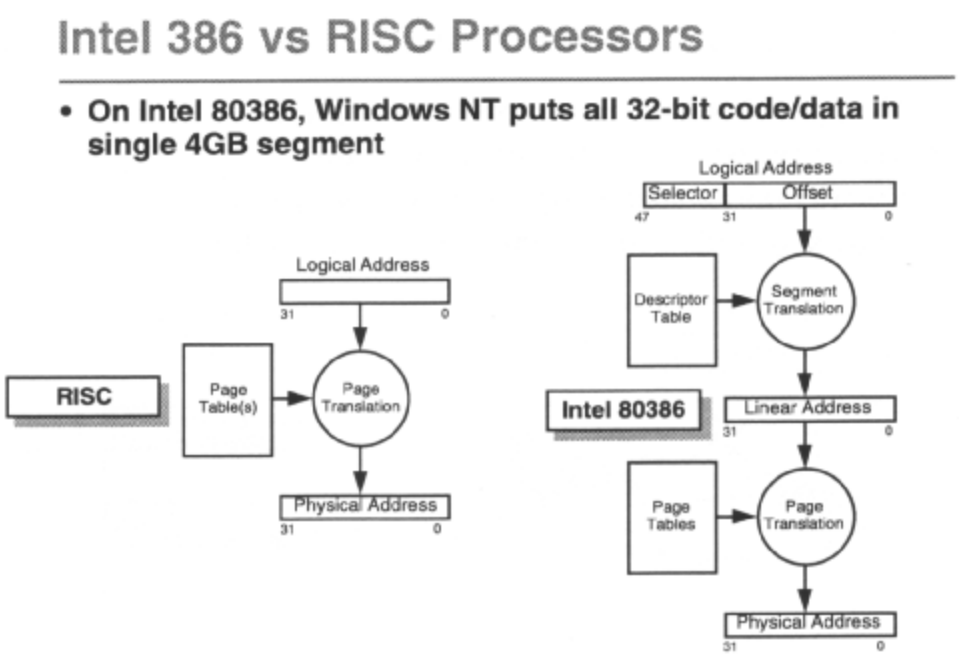
The memory management API allows a program to allocate memory while it is executing, and provides a mechanism for sharing memory. It also permits blocks of memory to be sub-allocated and offers backward compatibility for 16-bit applications.

In this chapter, we learn how to use the memory management API to reserve, commit, share and sub-allocate memory regions.

Objectives

When you have completed this chapter, you should be able to:

Describe the differences between the Win 16 and Win32-bit memory management APIs
Explain the difference between reserving and committing memory
Write programs that use dynamic memory allocation
Allocate shared memory, anonymously or by name
Sub-allocate blocks of memory from a memory heap



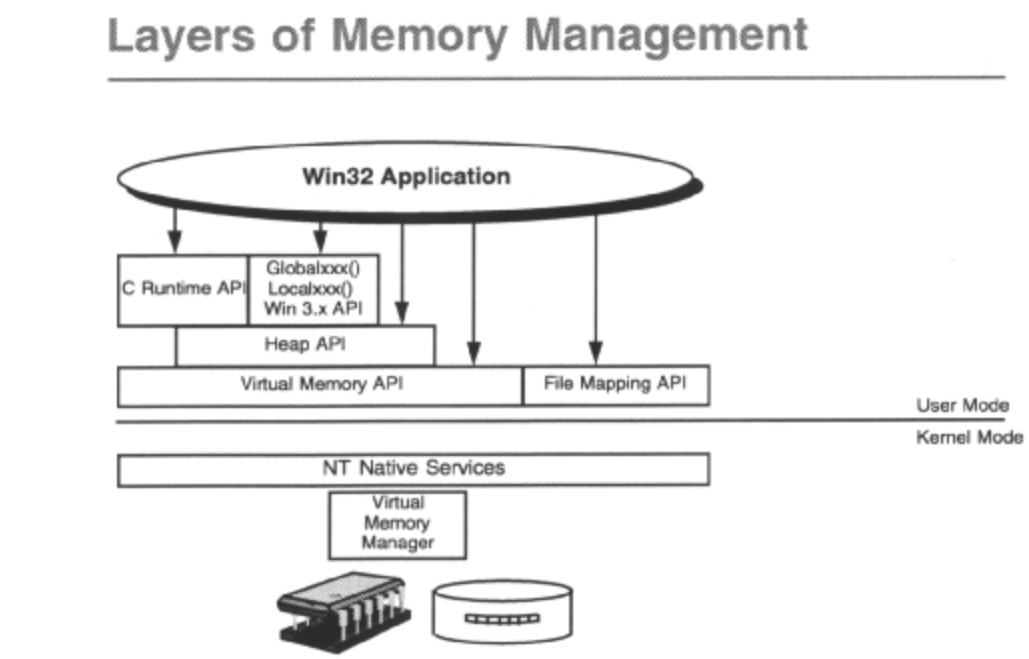
The Intel 80386 is a 32-bit processor with full backward compatibility with the 80286 and 8086. It is therefore capable of operating in three different modes: real, protected and paged-protected. In the paged-protected mode, a second level of address translation is introduced called paging. As the above diagram shows, in this mode, a logical address passes through two translation mechanisms: segmentation and paging.

Although the 80386 is a true 32-bit processor, it is capable of operating as a 16-bit processor, for example when it is emulating an 80286. Of course, it is much more powerful as a 32-bit processor because its registers are extended to 32-bits and can be used orthogonal. The 8086 and 80286 assign a specific function to many registers but the 80386 can use any 32-bit general register for addressing calculations and for the results of most arithmetic and logical operations. Performance is improved due to the use of 32-bit internal data paths and the support for larger segments, which can be as large as 4 GB. The 64 KB maximum size of 16-bit segments generally means that segment registers have to be frequently reloaded (with new selectors) which involves a performance hit in reloading segment descriptors from memory to hidden on-chip registers. The performance gains are even higher with the 80486 (a faster highly integrated version of the 80386) where due to very efficient pipelining, most core 32-bit instructions execute in one processor clock cycle (which is one definition of a RISC processor as we will see later).

One might question the need for both segmentation and paging but they are actually designed for different purposes. Segmentation is designed to provide a very large virtual address space (64 TB max) and excellent protection. On the other hand, paging is designed to support efficient swapping between physical memory and disk and to provide some protection. However, because (a) a 32-bit segment can be as large as 4 GB, and (b) the paging mechanism provides adequate protection, Windows NT, like UNIX and OS/2 2.0, places all 32-bit code in a single 4 GB segment. This results in the so-called “flat address” model where an application simply uses 32-bit offsets (i.e. near pointers in ‘C’) to address any byte within a 4 GB linear address space (as with any other non-Intel 32-bit microprocessor!). Note that both the Intel 80386 and 80486 fix the page size at 4 KB.

Windows NT is designed to be portable to any 32-bit (or 64-bit) processor with some form of paging mechanism. Today, Windows NT is available for the DEC Alpha AXP processor. In contrast to a CISC (Complex Instruction Set Computer) processor, such as the Intel 80386, RISC (Reduced Instruction Set Computer) processors, such as the Alpha, aim to optimize performance by eliminating on-chip micro code (thereby leaving more chip ‘real estate’ for efficient pipelining, etc.). The goal of a RISC design is to achieve an execution rate of one instruction per clock cycle (or better!). The distinction between CISC and RISC processors is becoming blurred as silicon technology allows more and more transistors to be placed on a single chip. Intel’s 80486 already includes some RISC -type features and the Pentium includes many more.

Do not assume that the page size is 4Kb on other processors. For example, it’s 8 KB on the DEC Alpha processor.



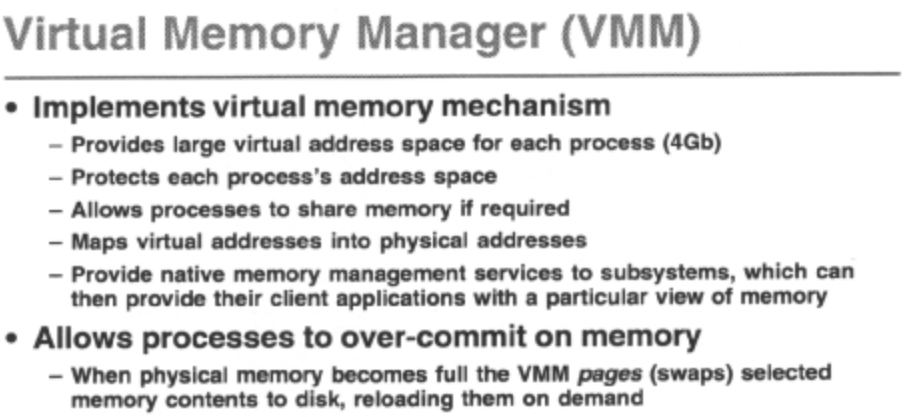
These are the Win32 memory management APIs we will look at. There are different types of API with different characteristics suitable for particular needs, some layered on top of others.

The virtual memory management API provides low-level control over the pages of the private address space of a process.

Memory mapped files functions provide the ability to fast file access and memory sharing. The Heap memory management API offers support for dynamic data structures, minimizing fragmentation.

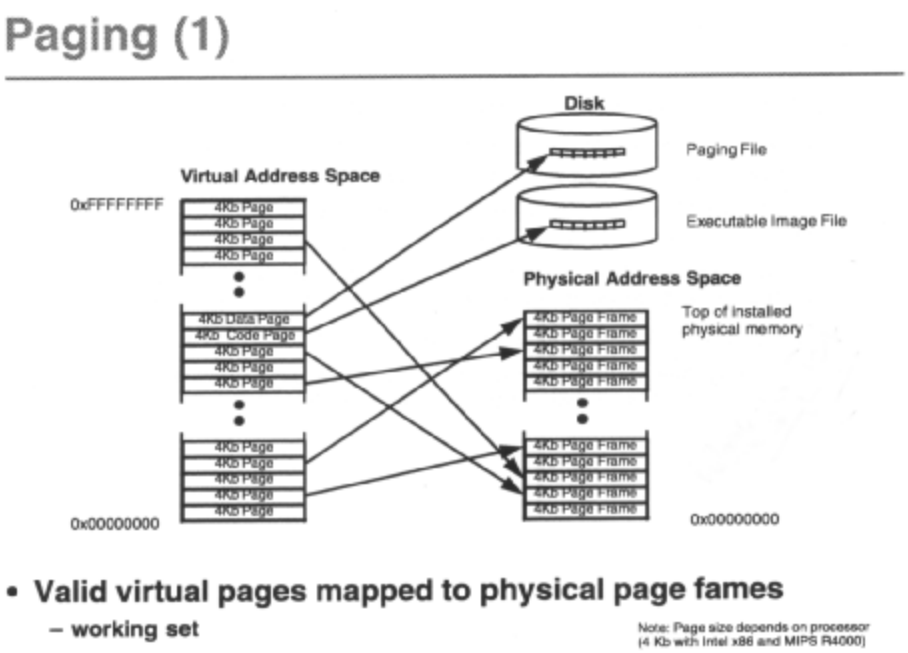
The C run-time functions provide portable dynamic memory capability, often built on the heap support provided to processes.

The old Windows 3.x `Globalxxx()` and `Localxxx()` memory management API are supported for backward compatibility and are built on the heap support provided to processes..



Like most modem operating systems, Windows NT supports *virtual memory.* Virtual memory is a logical view of memory that doesn't necessarily correspond to the underlying physical memory organization. At runtime, the operating system, with assistance from the processor hardware, translates, or *maps,* virtual addresses into physical addresses. Each process is given a virtual address space, which is usually much larger than the total physical memory available. When physical memory becomes full, the operating system *swaps,* or *pages,* selected memory contents to disk, freeing up memory for other uses. Pages swapped to disk are reloaded on demand. Because the operating system controls the mapping between virtual and physical memory, each process can be given a separate private virtual address space that is protected from other processes. Selected memory areas can also be shared between processes if required.

The implementation of virtual memory differs between operating systems, usually as a result of limitations of the host processor. 16-bit operating systems, like OS/2 I .x, that were originally targeted at the Intel 80286, use segmentation. Other 32-bit operating systems, like OS/2 2.0 and Unix, targeted at Intel 80386, Motorola 68K or **RISC** processors, provide a 32-bit (4 GB) *flat,* or linear, address space to each process. In the case of Windows NT, virtual memory management is more complicated because of the need to provide multiple operating system environments. Each environment subsystem therefore provides a logical view of memory that corresponds to what its applications expect. Underneath in the NT Executive, the Virtual Memory Manager views virtual memory as a flat 4Gb flat address space and provides a mechanism and a policy for implementing virtual memory and set of native memory management services for environment subsystems to call.



Because it is very inefficient to swap a byte at a time to and from disk, the virtual address space is divided up into blocks of equal sizes called *pages.* Similarly, physical memory is divided into blocks of the same size called *page frames.* The size of a page depends on the host processor. The Intel 80386, which provides comprehensive paging translation support in hardware, fixes the page size at 4 KB. The DEC Alpha chip fixes the page size at 8 KB, and the MIPS R4000 (not supported by NT 4.0 any more) supports programmable page sizes between 4 Kb and 64 Kb. Programmers shouldn't therefore assume the page size is 4 Kb!

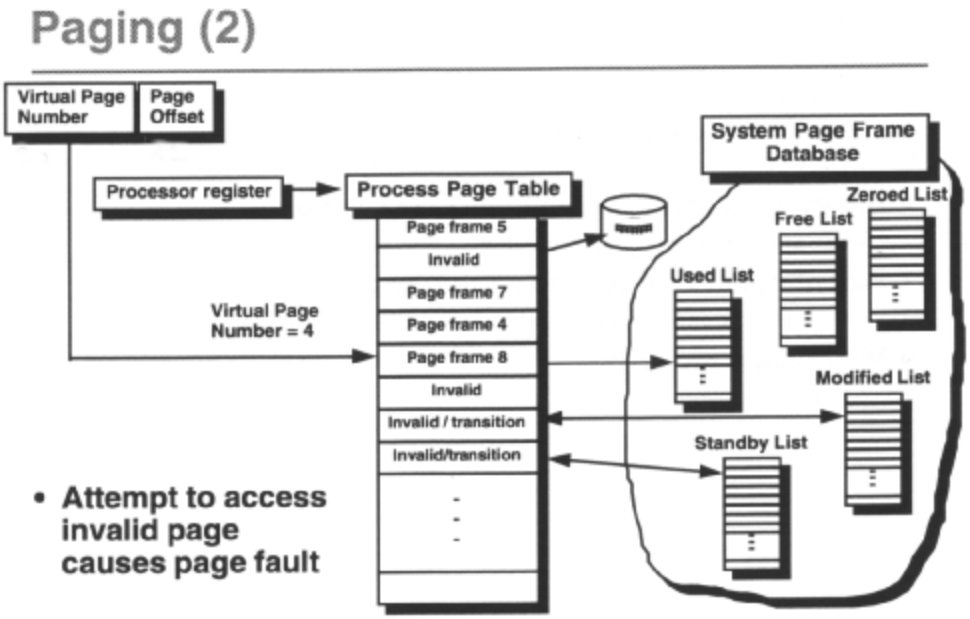
Any page in the virtual address space of a process can be in one of three states. If a page is mapped to a page frame in physical memory or to a page-sized entry in a file on disk it is said to be *committed.* If a page has been set aside for future use so that it cannot be re-used for any other purpose within the process, but has no associated physical page storage, it is said to be *reserved.* If a page is available to be reserved it is said to be *free.*

At any given time, a native process may only have a subset of *valid* pages in its virtual address space, i.e. those that are mapped to page frames in physical memory. This subset is known as the *working set.* Pages that are temporarily stored on disk or reserved or free are known as *invalid* pages.

When an executing thread attempts to access an invalid page, the processor raises a particular exception called a *page fault.* The VMM can respond in a number of ways - see a later slide on paging policy. To respond to a page fault caused by the invalid page being on disk, the VMM will load the required page from disk into a free page frame in physical memory and re-execute the instruction, which caused the fault. This activity is known as *demand page loading.* When the number of free page frames runs low, the VMM selects page frames to free and copies their contents to disk. This activity, known in Windows NT as *paging,* is transparent to the programmer.

The ability to map files into memory and swap pages of a *memory mapped file* between memory and disk provides the basis for paging all types of memory whether it is executable code, static data or dynamic data. Executable code and read-only resources are backed against their image file *(.exe* or *.dll)* on disk. The system-paging *file* is used to back data pages. Any arbitrary file can be mapped into memory. It should be clear that the amount of virtual memory available to applications is decided by the amount of physical memory plus, and more importantly, the available paging file space on disk. As long as hard disk space is available, paging files can grow dynamically when more space is needed.

Windows NT supports multiple paging files (up to 16), one on each logical disk, and can access several paging files simultaneously. Windows NT can also perform asynchronous I/O to the paging file.



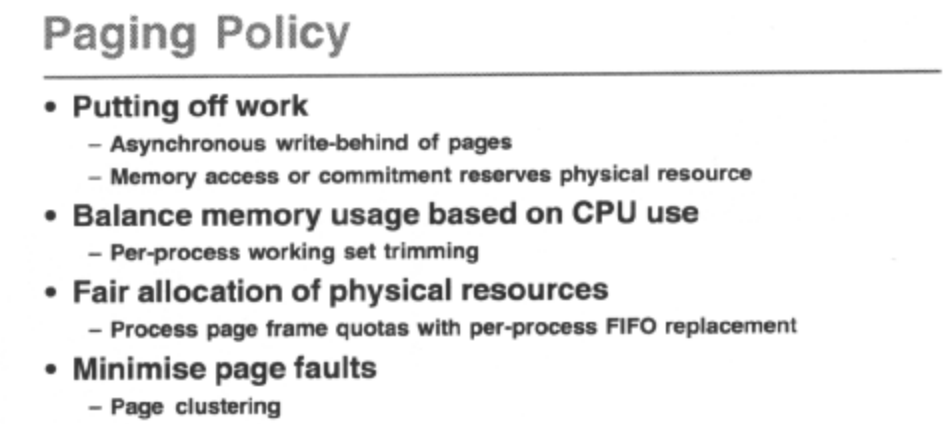
Mapping between the virtual and physical address spaces is achieved through *page tables.* Conceptually, the Virtual Memory Manager (VMM) builds and maintains s single page table for each process, as shown in the diagram above. (In practice, page tables are arranged in a two-level hierarchy to reduce the demands on physical memory). *Page table entries* in the page table describe each committed page in the address space of a process. As such a page table entry contains a reference to the page, in memory or on disk, and some attributes defining the page state and usage.
The page table address for a given process is mapped into a processor register, which it always uses to locate the page tables to translate addresses. Whenever a context switch occurs to schedule a thread of a different process then this register is remapped with the address of the new process page table, so protection between processes is natural.
One attribute of a page table entry specifies whether a page is *present,* or *valid,* or *not-present,* or *invalid.* If an entry is valid, it contains a physical page frame address. If an entry is invalid, it contains either a disk address of a page that is temporarily stored on disk or it can also be marked as *transitional.* An invalid transitional page is available for reuse by this or another process, but meanwhile it still contains a valid page frame address in memory. A valid page may have been written to since the last save to the backing file, *dirty,* or exists the same as when it was last read from its backing file, *not-dirty.* Pages also have other protection attributes, which are discussed later.

Details of each page frame in the system is stored in the *page frame database* and is marked as one of:

valid	pageframe in use, pointed to by valid page table entry
standby	pageframe still in RAM, page table entry invalid, but marked transitional *modified* like standby, but dirty
free	free, uninitialised
zeroed	free, initialized
bad	cannot be used

If an attempt is made to access a page that is invalid, the processor generates an exception, or *page fault.* The VMM responds by loading into memory, not only the page demanded, but also a small number of pages around it . This strategy, known as *clustering,* attempts to minimize the overall number of page faults, because page faults are expensive and the page size is relatively small. This also obviates the need for a dedicated disk caching mechanism.

Page faulting is the only way to get physical pages of memory loaded.
Of course if the page table entry is marked as transitional then pages can be made valid very quickly without a disk read.
Page table entries are 4 bytes wide, so on a system with 4Kb pages it takes 1024 page frames (2^20 pages*4bytes/4Kbytes) to describe one process address space! So even page table entries are paged to disk. Of course this means that more than one page fault can occur in a single address translation.



The VMM reduces the demands on physical memory by using what is called 'lazy' allocation. This means that **it** postpones expensive operations until the last possible moment. For instance, reservation requests are noted but physical page frame storage is not committed until memory is accessed or specifically committed. Regions of reserved memory are recorded in *VirtualAddress Descriptor* (VAD) structures and page table entries are not allocated until they are needed.

When physical memory is full and a page fault occurs, the VMM has to choose a page frame to remove and perhaps swap to disk. Unlike other operating systems, such as OS/2, Windows NT enforces page replacement on a per-process basis. Rather than a process causing arbitrary pages to be swapped to disk to satisfy it's memory commitment, the VMM will only swap valid pages from those of the faulting process. This ensures that memory hungry processes won't starve others, and thus adversely affect the system. A simple FIFO algorithm is used to select the page that has been in memory the longest.

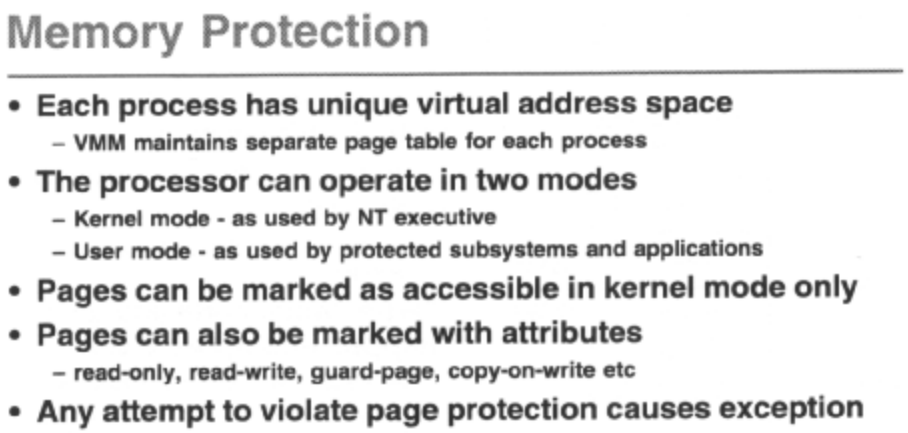
This means that the VMM must keep track of the valid pages each process is using, the process ***working set.*** Each process has a minimum working set size, the number of guaranteed pages in memory, and a maximum working set size, the maximum number of physical page frames which each process is allowed to use. NT also enforces a page file ***quota, which*** dictates the maximum space in the paging file, which each process is allowed to use.

When physical memory is not full the VMM allows a process to have as many pages as its working set maximum and perhaps more. If a process needs memory, which is initialized, page frames are taken from the zeroed list, then the free list (after zeroing them). If a process needs memory not necessarily initialized, page frames are taken from the free list, then the zeroed list. If either of these lists is empty, page frames are taken from the standby list.

However, asynchronous background threads work during system idle time to manage the system wide ***page pool*** in order to distribute memory fairly to all processes in the system, and maintain a sensible balance between free and used memory. As the free, zeroed and standby lists get low over time, the ***modified page writer*** thread of the VMM wakes up, writes modified page frames to disk and adds them to the standby list.

Also, overtime, the VMM slowly steals pages from the working sets of processes to increase the amount of free memory in the system. It does this to ensure that there is enough memory to satisfy a sudden demand for pages, and so that active processes get more memory than inactive ones. This is called ***automatic working set trimming.*** For each process using more pages than it's minimum working set size, the VMM slowly removes pages from the working set and puts them on the modified list. It then monitors the frequency of page faults, for each process that has reached its working set minimum, to judge how it has been affected and to decide how to further amend it's working set.

Remember that page frames that are on the standby or modified lists are pointed at by an invalid, transitional, page table entries. If such an invalid transitional page is referenced, it can be validated very quickly without reading from disk.



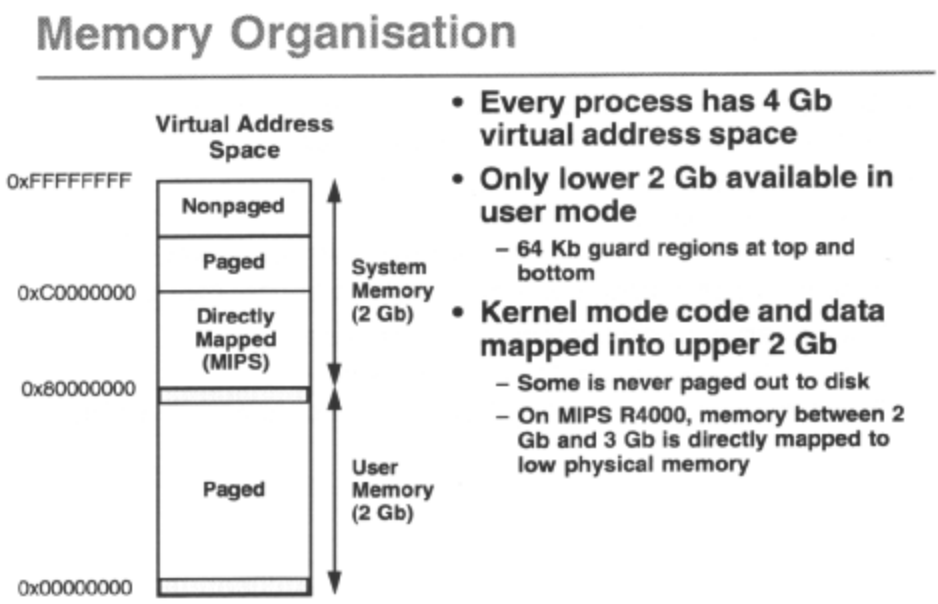
Windows NT protects virtual memory addresses using whatever hardware support is available in the host processor. A separate virtual address space is assigned to each process by the VMM, which maintains a separate page table for each process. Each page table maps the private address space of a process either to unique physical page frames, or to unique regions in a disk file. Whenever a thread is preempted by a thread in a different process, the Virtual Memory Manager (VMM) switches page tables. If an executing thread in a process attempts to access an invalid page, the processor raises an exception.

All 32-bit processors provide a mechanism for protecting system code. For example, the Intel 80386 supports the concept of privilege levels. At any point in time, a thread's privilege corresponds to the privilege level of the page of code it is executing. A page can be marked with one of two privilege levels; *user* or *supervisor.* An executing thread can only access or use a page with equal or lower privilege. Certain privileged processor instructions can only be executed by a thread running at supervisor level. Any attempt to violate these rules causes the processor to raise an exception. The processor provides a controlled mechanism for a less privileged piece of code to call a more privileged one. In Windows NT on an Intel x86-based system, kernel mode and user mode correspond to supervisor privilege and user privilege, respectively.

32-bit processors vary in their support of other page protection attributes. For example, the MIPS 4000 allows a valid page (one that is mapped to a physical page frame) to be marked as read-only or read/write. The processor raises an exception if a thread attempts to write to a read-only page. Page protection support in the Intel 80386 is similar.

Windows NT supplements these basic protection mechanisms, *read-only, read-write* and *no-access,* in software. The VMM also allows a page to be marked as *execute-only* (if the processor supports it), *guard-page, copy-on-write* and *non-cached.* The Intel 80386/486 doesn't support execute-only protection, so on these processors, execute-only access are equivalent to read-only access. The VMM uses the guard-page attribute to provide bounds checking on data items. This is especially useful to automate the growth of stacks. When a thread accesses a guard page at the top of the stack, the system generates an exception, which the VMM handles by growing the stack and providing a new guard page, if the stack limit has not been reached. The copy-on-write attribute is discussed later.

The Win32 subsystem makes a subset of the VMM' s page protection available to Win32 applications through its API.



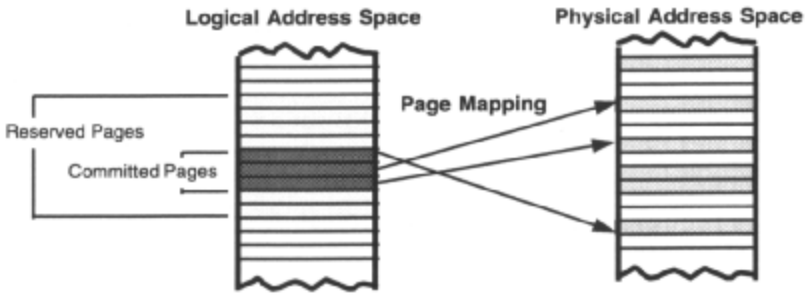
Every native Windows NT process has a 4 GB virtual address space, divided in half. The lower half is available to both user-mode and kernel-mode code and is unique to each process. 64 KB ‘no-mans land’ areas are provided at the top and bottom of this region to catch stray pointers and the like. All user mode process and DLL code, data and resources are mapped into this region including all the Win32 system code.

The upper half is reserved for kernel-mode code and is identical for every process. In other words, the NT Executive is mapped into the upper half of the virtual address space of every process. Any attempt by user-mode code to access any byte in the upper half will result in an access violation exception. 32-bit processors provide special mechanisms to allow user-mode code to call pre-defined entry points in kernel-mode code. The need to reserve 2 GB for the NT Executive is a limitation of the MIPS R4000 architecture. With other processors, such as the Intel 80386, suitable programming of segment descriptors could reduce the system memory region, but 2 GB is always reserved to ensure portability.

Part of the kernel-mode code and data resides between 2 and 3 GB and is never paged out of physical memory. Other parts of the NT Executive reside in the region between 3 and 4 GB, part of which is reserved for code and data that can never be paged to disk (e.g. the paging code itself!).

Environment subsystems present views of memory to their client applications that do not necessarily correspond to the virtual address space of a native Windows NT process. For example, the I 6-bit OS/2, Virtual DOS Machine (VDM) and Windows on Win32(WOW) subsystems present very different views. However, the Win32 and POSIX subsystems present their respective client processes with views of virtual memory that are identical to that of a native Windows NT process.

Reserving and Committing Memory



- **When a memory region is reserved, one or more contiguous pages are set aside in logical address space**
- **Only committed pages are mapped to physical memory or to disk space**

The Win32 subsystem allows memory to be allocated in two steps: first the memory can be *reserved* and then it can be *committed.* These are fundamentally different operations.

A memory region is defined as a contiguous range of virtual addresses (a set of contiguous pages) that have identical state and protection characteristics and the same allocation base address.

Reserving memory sets aside a region in the virtual linear address space of the calling process, for its future use. Any subsequent request by any thread in the same process to reserve a memory region will not use reserved pages; reserved pages must be freed in order to be re-used.

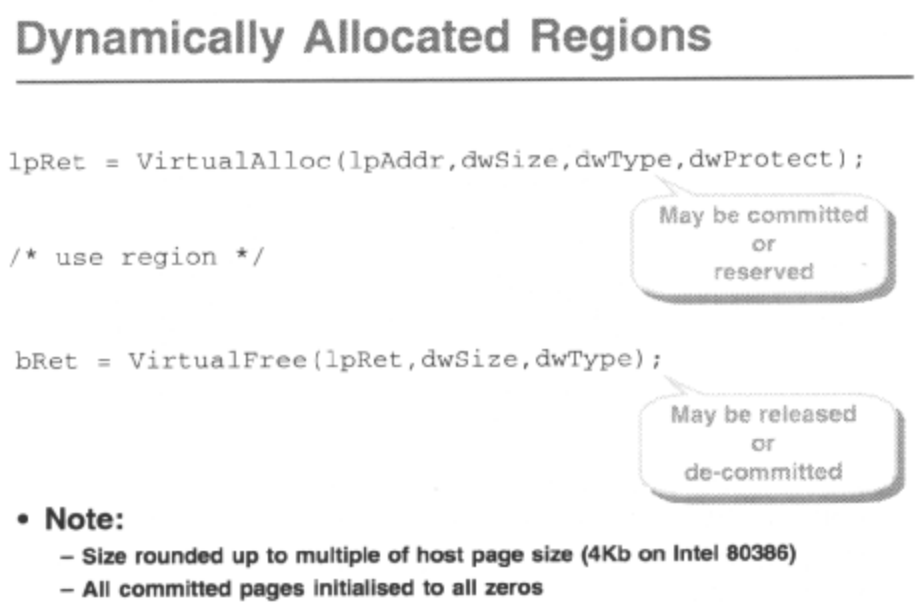
Reserving pages is very fast and takes up little system resource. Regions of reserved memory are recorded in *Virtual Address Descriptor* (VAD) structures held in a balanced B-tree and no page table entries are built until reserved pages are committed. Nor is physical memory space or disk space in the paging file actually demanded until reserved pages are committed.

Committing memory forces Windows NT to map one or more virtual memory pages within a region either to physical memory or to a paging file on disk. Memory may be committed a page at a time if required. Committed memory can be backed by any memory-mapped file - an image file, any arbitrary file or the system-paging file.

Committing memory is fast, like reserving addresses, since a page of physical memory is allocated only when page faults occur. Also, the backing store is allocated immediately, but no data is written to it until dirty pages become paged out of physical memory.

Reserving memory without commitment is useful for large dynamic data structures where an application needs to guarantee that a certain number of pages are available at the end of a committed region to facilitate linear growth. Once a region is reserved it cannot be re-sized. In the current implementation, the minimum range of addresses that can be reserved is 64Kb.

Each process has a quota of system resources it is allowed to use. This is to stop any one process from hogging a disproportionate amount of the system resources and adversely affecting other processes. One of the system resources controlled in this way is the paging file. Windows NT deducts from a process paging file quota for committed pages, but not for reserved pages.



A Win32 application can call `VirtualAlloc()` to reserve a region of free pages within its virtual address space by using the MEM_RESERVE flag. One or more pages within a region of reserved pages can be committed with VirtualAlloc() by using the MEM_COMMIT flag. A region of free pages can be reserved and committed by specifying both flags. Note that an attempt to access an uncommitted page will cause a page fault exception.

Once pages are committed, an *access protection* value can be assigned to them, calling VirtualProtect , but reserved pages are given a protection value of PAGE_NOACCESS. Initially committed pages are zeroed. Pages committed by `VirtualA11oc()` can be accessed (if they are not protected) using normal pointer references. Other memory allocation functions such as `GlobalAlloc()` and C runtime routines such as `malloc()` provide read-write committed memory.

One or more committed pages within a region of a calling process can be decommitted with `VirtualFree()` by using the MEM_DECOMMIT flag. All pages within a region of reserved or committed pages can be freed with `VirtualFree()` C by using the MEM_RELEASE flag.

Here is an example of how to allocate a committed, read-write memory object of 70Kb:

```
LPVOID ip =  VirtualAlloc( NULL, 71680, MEM_COMMIT PAGE_READWRITE );

VirtualFree( ip, 0, MEM_RELEASE);
```

Note that the actual size of the memory region allocated and committed is rounded up to a whole number of pages: 72Kb, or 18 pages, for instance, on an Intel 80386 processor. In the worst case, an object which is 1 byte longer than an exact multiple of a page size will waste an entire page of memory, less one byte, when its last page is physically present in memory. This is known as *internal fragmentation.* With small objects of 10Kb or less, the amount of memory wasted in this way becomes comparable with the size of the object itself. Heaps often provides a more efficient way of allocating many small blocks of memory.

Remember that Windows 3.x enforces memory protection on segment boundaries, and programmers often use general protection faults to trap bugs caused by invalid pointers. This technique is no longer valid for Windows NT, however, because the flat memory model places all code and data in a single segment. Page protection is enforced on page boundaries, therefore a page-fault violation won't occur unless an attempt is made to reference an invalid page.

Sparse Memory Regions

- **Once allocated, memory regions cannot be re-sized**
  - **Individual pages can be committed or de-committed**
- **Memory region should be reserved (but not committed) for "worst case" maximum size**
  - `VirtualAlloc(), VirtualFree()` **should then be used to commit and de-commit pages as required**
- **Exception handler can be used to automate commitment of pages**

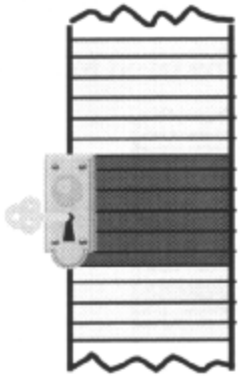
The size of a 16-bit segment may be changed by a call to `GlobalRealloc()`, but once a 32-bit memory region has been reserved, its size is fixed.

In the segmented memory model of the 16-bit API, the memory manager can change the size of a segment by updating the limit field in its descriptor. This is not possible in the flat memory model because a memory region consists of one or more pages of fixed size, which must be contiguous in the logical address space. It is possible to pre-allocate very large memory regions, however, because the reserving of uncommitted pages does not in itself demand pages in physical memory or in the swap file on the hard disk. A memory region with one or more uncommitted pages is known as a *'sparse'* memory region.

Individual pages in a sparse memory region can be committed as required by using `VirtualAlloc()`. An application can also check whether a page is committed or uncommitted by calling `VirtualQuery()` As will be explained, it is possible to automate this procedure by providing an appropriate exception handler.

Locking Pages

- `VirtualLock()` **and** `VirtualUnlock()`
- **Prevents pages being swapped from working set to paging file**
- **Limit imposed by Win32 to prevent excessive performance degradation**



Pages within a specified region of a process virtual address space can be locked into memory, using `VirtualLock()`, ensuring that subsequent access to the region will not incur a page fault. Of course, all pages in the specified region must be committed to be locked.

Note that the actual size of the memory region locked is rounded up to a whole number of pages: in the worst case, a region to be locked which straddles two pages by 1 byte each will lock both pages

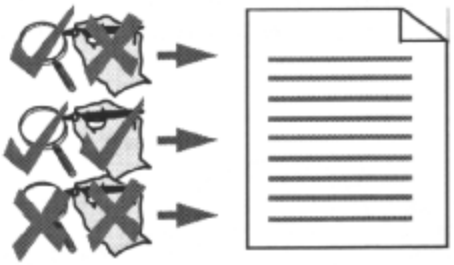
`VirtualUnlock()` unlocks pages. `VirtualLock()` does not increment a lock count so only one call to is `VirtualUnlock()` required to unlock a region of pages. Locked pages are automatically unlocked when a process terminates.

NB. `VirtualLock()` only locks pages into a process's working set ensuring they don't become swapped as part of the local HFO replacement policy; it does not lock them absolutely into physical memory. Therefore as long as a given process is in physical memory, the virtually locked page is guaranteed to be in physical memory as well. However, the system is free to swap out any virtually locked pages if it swaps out the whole process. When the system swaps the process back in, the virtually locked pages may end up mapped to different physical page frames.

Note also that it is wise to use `VirtualLock()` very sparingly because it reduces the flexibility of the system; locking pages into memory may degrade system performance by reducing available physical RAM and forcing the system to swap out other critical pages to the paging file. Therefore a limit of 30 locked pages per-process has been enforced. Depending upon memory demands on the system, the VMM may vary the number of pages a process can lock.

Access Protection

- **Committed pages may be given any protection status**
  - PAGE_READONLY
    - **Read access is allowed**
  - PAGE_READWRITE
    - **Write access is allowed**
  - PAGE_NOACCESS
    - **No access is allowed**



- VirtualProtect() **changes protection**
- **Reserved and free pages have a** PAGE_NOACCESS **protection status**

It is possible to query or modify the access protection of any page in the process's virtual address space. This allows you, for example, to allocate read/write pages to store sensitive data, and then change the access to read-only or no access to protect against accidental overwriting.

The access to each reserved or free page is determined by the following access right:

no-access No access to the pages is allowed. An attempt to read, write, or execute the page results in an access violation. PAGE_NOACCESS.

The access to each committed page is determined by its access right as follows:

read-only Read access to the page is allowed. An attempt to write the page results in an access violation. PAGE_READONLY. May be used the way discardable segments of memory are used in Windows 3.1. They are never 'dirty', so the system may use them (zeroing them first if necessary) without having to first write their contents to disk. From a programming standpoint though, when attempting to access a 'discarded' page, a page fault is generated, and the system reads it back in.

read-write Read and write access to the page is allowed. PAGE_READWRITE.

guard-page Any access to the page causes a *guard page entered* exception to be raised in the subject process. This is useful for implementing dynamically sizeable data areas. PAGE_GUARD.

no-cache Page will not be cached so that changes are immediately written to the backing store. PAGE_CACHE.

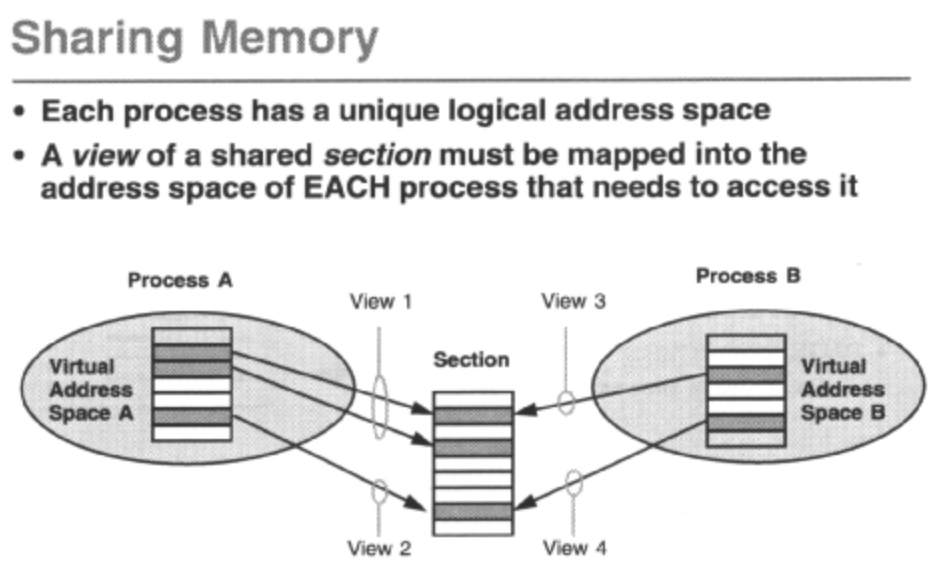
copy-on-write Used, mostly by the system, to share memory. Effectively a read-write page which references a page frame which may be shared by many processes. Any write access to the page causes the system to duplicate the page frame and mark the page as read-write. PAGE_WRITECOPY.

execute-only Execute access to the pages is allowed. Read or write access results in an access violation PAGE_EXECUTE. **There are variations on** this theme, namely

PAGE_EXECUTE READ, PAGE_EXECUTE_READWRITE, and PAGE_EXECUTE_WRITECOPY.

These access rights can be set up for a region of committed pages with `VirtualAlloc()`, or can be changed subsequently with `VirtualProtect()`. Freeing and de-committing pages, or reserving pages will give a page-level access right of PAGE_NOACCESS. An application can also check on the access protection of a page by calling VirtualQuery().

With the correct security clearance, `VirtualProtectEx()` and `VirtualQueryEx()` can be used to set/query page protection in another process. Note that the access protection attribute applies to pages. Don't assume page size is 4Kb. Use `GetSystemInfo()` to find the machine host page size dynamically.



Whilst virtual memory provides each process with its own private logical address space, ensuring that other processes cannot have access to it, virtual memory also provides a neat mechanism for sharing memory.

The Windows NT approach to sharing resources is to define them as objects. This is what it does with memory. The NT Executive *section object,* which is the basis for the *Win32file mapping object,* represents a file on disk, which potentially can be mapped into the address space of any process.

A portion, or *view,* of the file represented by the file mapping object can be mapped into the address spaces of multiple processes in such a way that the, potentially different, virtual addresses in all processes refer to the same physical page frames. Thus processes can share memory. Different processes can map the same views, different views, or multiple views, of the same file. Whilst a view may map the whole of a file, views obviously conserve virtual address space by allowing only sparse portions of the file to be mapped (committed).

Processes must be careful to synchronize their access to shared data if at least one process has write access.

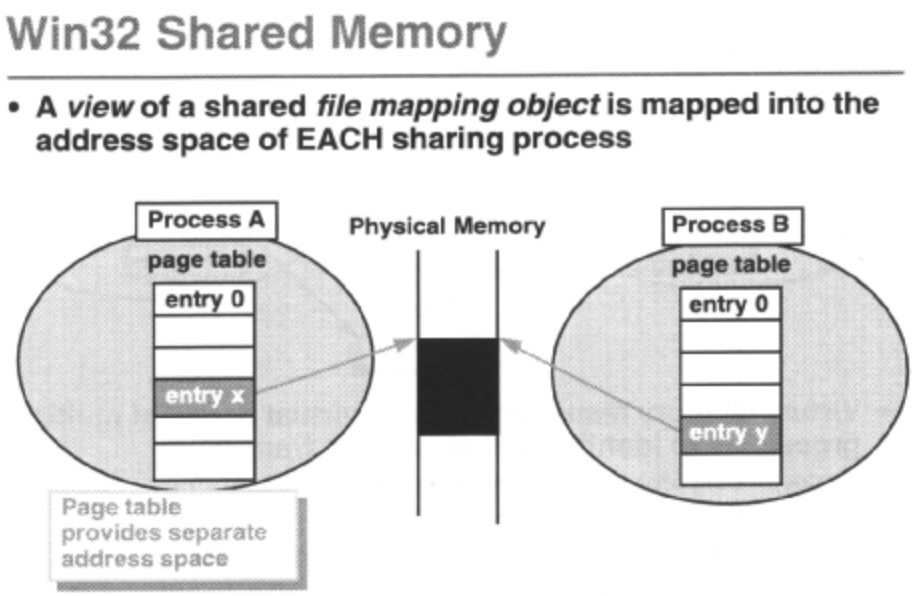
All system paging files, available for all processes to use, are memory mapped. The Executive however, allows pages in a section object to be mapped to a specified file in the filing system. This can be any file on disk, assuming it is open with the correct access, or it can be one of the system paging files. The paging to and from a specified file is transparent to the programmer, because the VMM uses exactly the same policy and mechanism as it does for paging to and from the system paging files. It just uses a different file.

Sharing memory is useful enough, but this technique can simply be used to speed up random or sequential **I/O** file operations, as the file is treated as a large array in memory.

Other than DDE, memory mapped files offer the only mechanism for sharing memory in Win32. A DLL can share data between processes that load it, but internally this data sharing relies on the same technique as memory mapped files.

This technique is also used when Windows NT allows two processes to share physical memory to save system resources, by using *'copy-on-write'* pages, which are then duplicated should one of the processes write to the memory. The two can share the same copy-on-write pages until one process writes to the memory. In this case the affected page frames are duplicated with *'read-write'* access, and the page table entries of the writing process are changed to point at the duplicates, so the physical memory is no longer shared.

All code and static data, which is common to processes, is by default shared, using copy-on-write page protection, providing a mechanism for multiple instances of an application without redundant image data. This is why loading second instances is quicker since the code and unwritten data page frames exist in physical memory already.



Functions such as `VirtualAlloc()`, `GlobalAlloc()` and `malloc()` dynamically allocate a private memory region. This is only accessible to threads in the process that issued the call. Each process has a unique logical address space. Note that dynamic memory allocated by DLL code is in the address space of the process that invoked the DLL, and is not accessible by other processes using the same DLL. A shared memory region is one that maps into the logical address space of two or more processes.

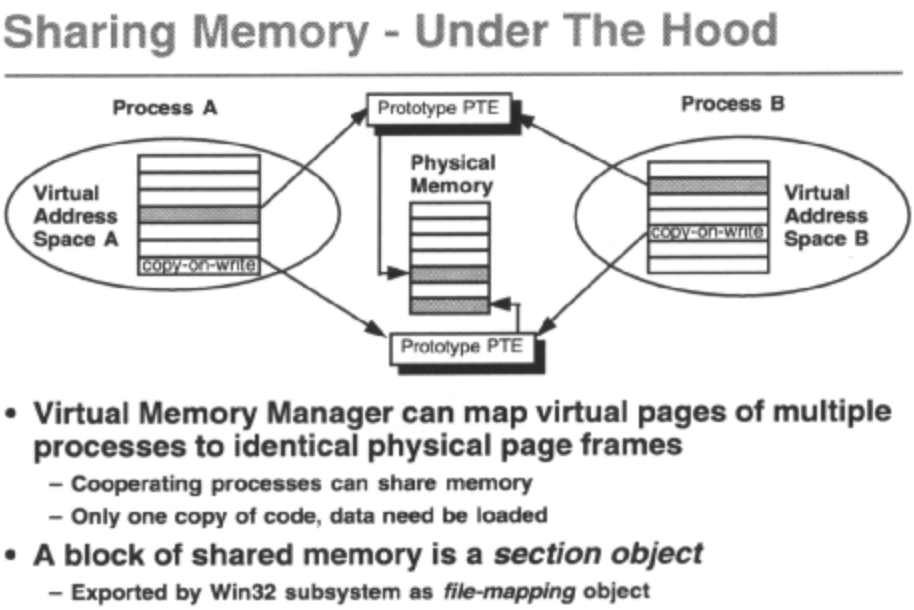
Note that use of `GlobalAlloc()` with the GMEM_DDESHARE flag will access private memory. However, passing a memory handle so obtained with a WN_DDE_XXX message or to the clipboard, will result in Win32 passing shared memory between 16 and 32-bit applications. This is for compatibility reasons. The preferred option for DDE is to use the DDEML with it's own mechanism for creating shared DDE memory objects.

A file-mapping object is normally created by name by one process, so that other processes can also open it by name. A default protection is specified for any pages, which are subsequently mapped. The file-mapping object can refer to a specific file in the file system or the system paging file. Creating or opening the object returns a handle to it. Once a process has the handle, it can map all or parts of the file represented by the file-mapping object into its own address space, specifying desired access to the region. Any attempt to open the file mapping object, or map it, in a way which would violate the default page protection specified at the time the file mapping object was created, will fail. Any attempt to change the protection on pages within a mapped view is subject to the same constraint.

Because shared memory is implemented by using an executive object, it gets an extra level of protection, namely the security validation applied to all objects uniformly. Once an open handle has been successfully obtained, the pages it references are subject to usual page access protection. The protection of pages within a mapped view cannot be changed if the change violates the security descriptor for the file-mapping object.

Multiple views of a file are guaranteed to be coherent unless the file is being accessed by normal file I/O functions, or unless the file being mapped is remote.

Generally the same view of a file will be mapped at different virtual addresses in different processes. However, views of a file may be mapped at the same virtual base address in all sharing processes, although this assumes that the same virtual region is available in all processes. This makes it possible to share pointers between processes. However the calling process can only suggest an address to be used to map the file at. If the required region is not available then the mapping will fail.



The *Virtual Memory Manager* (VMM) provides each process with a private unique address space by using separate page tables for each process. However, it is sometimes desirable to share memory between two or more processes.

For example, a number of processes may need to share a large database. Also, if a user runs two copies of a large program in different windows, it is clearly inefficient for Windows NT to have two identical copies of the same executable image in memory. The VMM easily accommodates these needs by simply mapping the pages that need to be shared into the virtual address space of the relevant processes, as shown in the diagram above.

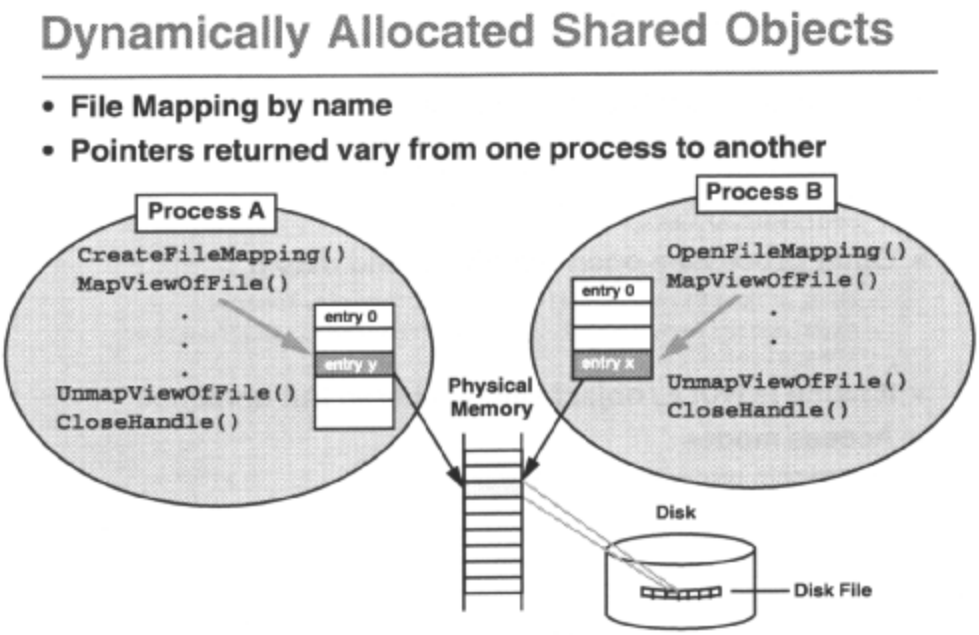
Prototype page table entries are used to implement shared memory. They look like normal page table entry, but they provide another level of indirection by pointing to a page in a private system data structure instead. Since the system data structure resides in the upper 2Gb of the address space for all processes, each process addresses it identically regardless of page directory. An unlimited number of processes can address the same prototype page table entry, which in turn identifies a specific page frame that is shared by all processes. This allows the sharing of page frames without the VMM needing to update the pages in the address space of every sharing process.

Like all shared resources in Windows NT, shared memory is implemented as an object. The Executive *section object,* which the Win32 subsystem makes available as a *file-mapping object,* represents a memory-mapped file that two or more processes can share. A section object can be backed up by the paging file, an executable image file or any other arbitrary file. The VMM automatically pages to and from the file, loading pages from disk when they are needed, and writing pages to disk when they are modified. Mapped files are used by the NT Executive to load executable images into memory and to implement automatic disk caching. A Win32 program can also explicitly use *mapped-file I/O,* accessing a file like a block of memory.

Since a section object can potentially be very large, it is possible for a process to map (commit) only a subset or *view* of a section into its virtual address space. A view provides a “window” into the section object, which can be moved as required. Memory sharing is simple since any process can access a file-mapping object if it has an appropriate name or handle.

The VMM also saves memory by using *copy-on-write* virtual page protection to share read/write pages between processes, as long as none of the processes write to them.
The processes all share the same physical page frames. As soon as a process writes to a copy-on-write page, the shared physical page frames must not be affected, so the VMM makes a copy of the physical page frame and makes the page being written to, reference it with read-write protection. Now the writing process has its own physical page frame and all other processes can go on sharing the original physical page frames.

For example, Win32 uses this protection to implement static data in applications and per-process static data in DLLs. Multiple invocations of the same application share static data pages until they are written to. This feature is also used
in the POSIX subsystem to mark code and data in a `fork()` operation. Code can be also marked copy
-on-write so that debuggers can write breakpoints to an application being debugged without affecting any other invocations of the application which are not being debugged.



File mapping provides a way for applications to share memory. It enables applications to view me contents of memory or a file (or a section of a file) as part of their address space, if they have the right name or handle.

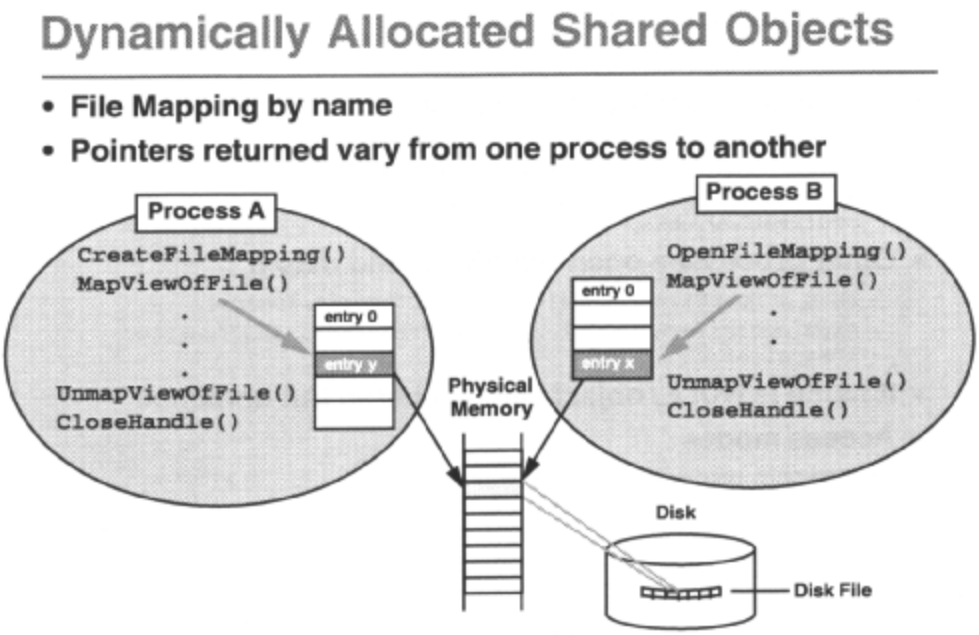
A file mapping object is created with `CreateFileMapping()` , specifying a file handle, which returns a handle to the mapping object. If the file handle QxFFFFFFFF is specified, then the system-paging file is used. The file-mapping object is referred to by this handle, which is used to ***map a view*** of the file. This handle can be inherited by child processes or communicated to other processes to be duplicated. Alternatively, a name can be associated with a file-mapping object when it is created. This name can then be used by other processes to obtain a handle to the object.

To implement shared memory, each process must use a handle to the same file mapping object to map a view of the file into its virtual address space. Typically, one process creates the mapping object with `CreateFileMapping()` and then maps the object into its own address space using `MapViewOfFile()` Other processes can then open the file mapping object with `OpenFileMapping()` specifying a name or using the mapping object file handle, and map the object into their own address space using `MapViewOfFile()` . Note that the virtual addresses resulting from the mapping in each process can vary, although the physical storage referenced is the same, shareable, memory. This shareable memory is only visible to processes that have mapped it into their address space. Once mapped, memory can be accessed according to specified access rights.

The name used for a file mapping exists in its own name space and will not conflict with other named objects like synchronization objects. It can be NULL for an unnamed file mapping object, and can't contain the "\" character. If a file mapping object is created from an existing file, created with `CreateFile()` , then it should be opened with exclusive access , to prevent other processes from accessing the file, and kept open until the memory is finished with.

When a process has finished using a file mapping object, it should unmap it by calling `UnmapViewOfFile()` which invalidates the shared memory for that process and flushes any changes to the file (if it exists). Flushing to file can be done at any other time by calling `FlushViewOfFile()`. Note that when flushing a memory-mapped file over a network, `FlushViewOfFile()` guarantees that the data has been written from the workstation, but not that the data resides on the remote disk. This is because the server may be caching the data on the remote end. Therefore, `FlushViewOfFile()` may return before the data has been physically written to disk. However, if the file is created with FILE_FLAG_WRITE_THROUGH, the remote file system will not perform lazy writes on the file, and `FlushViewOfFile()` will return when the actual physical write is complete.

Note also that `MapViewOfFile()` increments a usage counter that is decremented by `UnmapViewOfFile()` or on process termination. The physical named object is not removed from memory until the last process has unmapped the object, and the usage counter is set to zero.



Read-only, read-write or copy-on-write access can be specified for the mapping object while it is mapped, at the time it is created, and an optional security descriptor specified. When the mapping object is opened by another process, it specifies a desired access mode, which is checked against the security descriptor for that object. When a mapping object is mapped, a desired access mode is specified, and checked against the granted access rights to the object, which results in granted access rights to the shared memory pages (or an access denied error). File mapping provides committed pages. The inheritance flag on opening a mapping object specifies whether this handle can be inherited by related processes, and is checked against the security descriptor for the object.

Note that if a NULL name is specified for the mapping object, it can still be shared with other processes, by means of handle duplication and/or inheritance. This is more complicated as it may mean passing the handle between processes using some **IPC** mechanism.

Synchronization objects may be needed to prevent one process trying to open a mapping before another has created it, and to prevent two processes with write access to the shared memory, from simultaneously updating the object.

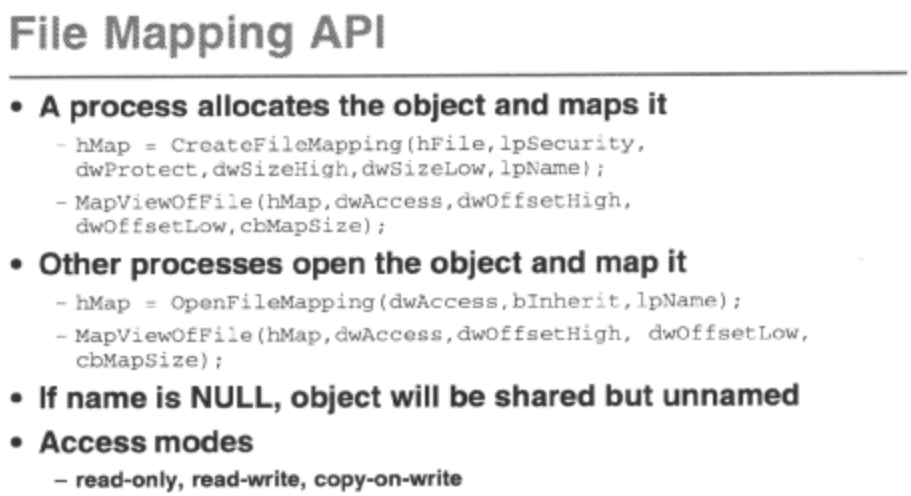
Here is an example of using 8Kb of shared named memory:

```
/*server process*/

char *pch;
HANDLE hMap=CreateFileMapping(OxFFFFFFFF, NULL, PAGE READWRIEE, 0, 0x2000,
“shrmem”);

if (hMap && GetLastError()= =ERROR_ALREADY_EXISTS)
{
      CloseHandle (hMap);
      hMap=NULL;
}
if (hMap)
      pch = (char *)MapViewOfFile ( hMap,
                              FILE_ MAP_READ | FILE_MAP_WRITE, 0 , 0 , 0 ) ;

/*client process*/
HANDLE hMap=OpenFileMapping (FILE_MAP_WRITE, 0 “shr_mem”);
if (hMap)
      pch = (char *) MapViewOfFile (hMap,
                              FILE_MAP_READ | FILE_MAP_WRITE, 0 , 0 , 0 ) ;
```



Ordinarily under Win32, it is not possible to share the block of memory by passing the address or global handles from one process to another. Both of these methods for sharing data work in Windows 3.x because all processes run in the same address space, but fail under Win32 because each process has its own address space.

Memory-mapped files are the best technique for sharing memory between applications (or shared data in a DLL covered later).

Some window messages specify the address of a block of memory in their iParam parameter. For example, the WM_SETTEXT message uses the `lParam` as a pointer to a string that identifies the new text for the window. If you send this message to a window in another process it will work even if the address you specify refers to private memory in your address space.

This is because Win32 traps the sending of WM SETTEXT messages and repackages the contents of the text into a block of memory to be shared with the other process. Special processing like this has to be performed for any inter-process message whose parameters represent a pointer. What about user defined messages like `(WM_USER + i)` that the system doesn't know about?

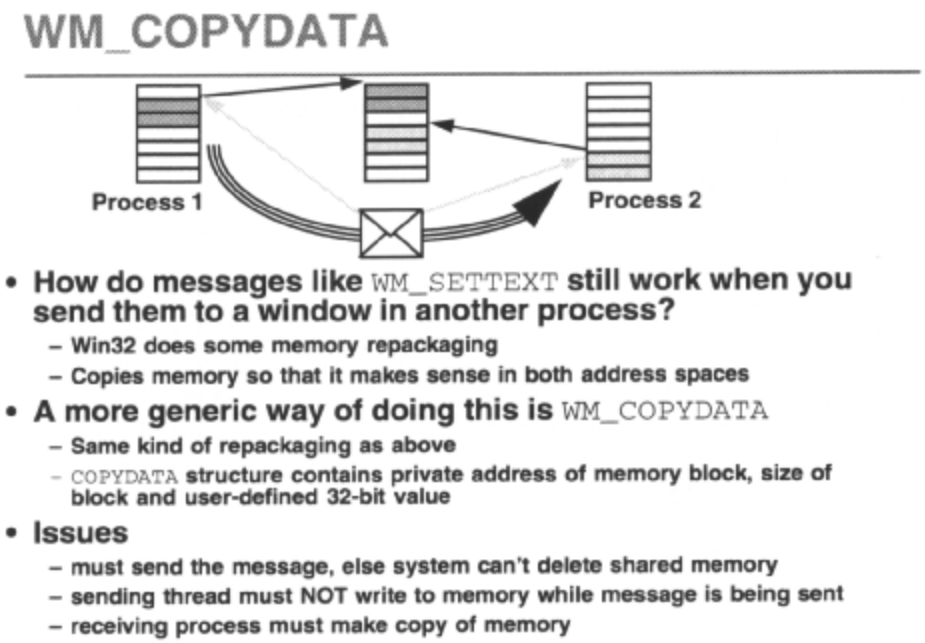
If you want to send a message which passes an address use the WM_COPYDATA message which takes a pointer to a COPYDATASTRUCT structure. It specifies the private address of the memory you want to share, the size of the memory and a 32-bit user-defined value which could be used to indicate the content of the data you are sending.

Similarly to the way it deals with WM_SETTEXT, Win32 traps the sending of a WM_COPYDATA message and repackages the contents of the memory into a block of memory to be shared with the other process.

You must send a WM_COPYDATA message because the system must free the copied memory after the receiving window procedure has processed the message. If you post the message, the system doesn't know when the WM_COPYDATA message is processed.

It takes some time for the system to repackage the data into the other process's address space which means that other thread running in the sending application process must not modifies the contents of the memory block during the period until the `SendMessage()` returns.

The receiving application must make a copy of the shared memory it is given during the processing of the WM_COPYDATA message. This is because the system deletes the memory after the message is processed.

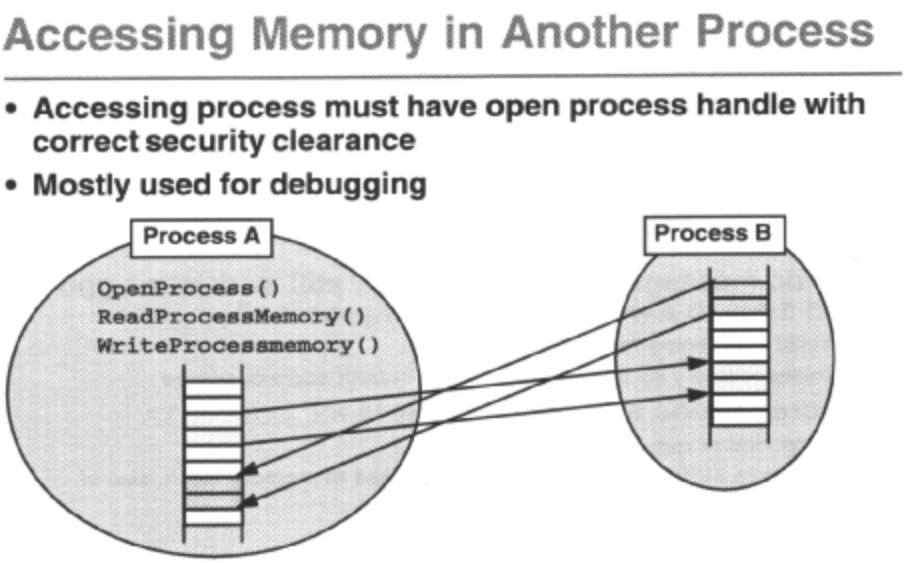


The `WriteProcessMemory() and ReadProcessMemory()` functions allow one process `to` write to and read from virtual memory in a specified process, as long as security permits. The entire area to be written to/read from must be accessible, or the operation fails.

`VirtualProtectEx()` and `VirtualQueryEx()` allow one process to set/query page protection in the virtual memory of a specified process, as long as security permits.

Any process that has a handle to another with PROCESS_VM_WRITE and PROCESS_VM_OPERATION access can call `WriteProcessMemory()` and any process that has a handle to another with PROCESS_VM_READ access can call ReadProcessMemory() . The process whose address space is being written to/read from is typically, but not necessarily, being debugged.

`OpenProcess()` enables one process to open another, and obtain a process handle, with the necessary access. `CreateProcess()` enables one process to start another, and obtain a process handle, with the necessary access.



Win32 provides high-level memory management functions enabling applications to manipulate dynamically allocated memory, and which are compatible with heaps implemented in Windows 3.x. Applications can create multiple, private heaps whose primary function is to efficiently manage the dynamic memory requirements of a process.

Heaps are independent regions of virtual memory, identified by handles. Blocks of memory, identified by pointers, are sub-allocated from a heap at run time and managed by a heap. Heaps simplify the finer details of system resources during memory management, such as the difference between reserved, free, and committed memory, and also provide smaller allocation granularity than the virtual memory management functions, so avoiding the internal fragmentation that can result from the allocation of many small objects. Without the heap manager, applications would be forced to use the virtual memory management functions, which allocate a minimum of one page of memory at a time.

A heap spans a range of pages in virtual address space, some of which are reserved, others that are committed. The addresses are contiguous, and they all originate from the same base allocation address. The techniques they use to manage memory use the available page-based virtual memory management functions. The heap manager searches for the first available block of memory that is large enough to satisfy any allocation request. If an allocation request cannot be satisfied from the committed region then the heap manager automatically, and transparently, commits pages of memory as needed to satisfy the request. When a block of memory is freed, the manager checks the previous and following block to see whether either or both can be combined with the newly freed block to form a contiguous free block. In either case freed memory is re-used if possible. Pages in the heap have read/write access and once committed, pages will not be decommitted on freeing memory blocks, but when the process terminates or the heap is destroyed.

Apart from normal paging, memory is not strictly ***'moveable'*** but can be re-allocated at a different virtual address under application control. Therefore heap fragmentation may occur over time if many small variable-size memory blocks are sub-allocated and freed. This is because once a block becomes free it may be the wrong size to re-allocate. It is perhaps more sensible to use multiple heaps, each storing data of a fixed size, to store data of variable sizes. This way the heap manager has no problems re-allocating blocks when they become free.

The heap manager requires additional memory from the heap to store heap support structures for managing the memory in the heap. There is one heap-specific overhead and an overhead for managing each sub-allocated block of memory.
The type of the heap, the type of the memory and the allocation size determine the extent of this overhead. Information such as block size, status (used or free) and a pointer to the next block, is kept in a header in the first few bytes at the beginning of the block, not in a separate location, such as a table.
Problems occur if you corrupt the header information of a block by writing beyond the previous allocated block. The blocks are contiguous, so overwriting one block when writing to a different block is not considered writing outside the programs memory space.
The heap manager doesn't make sure that writes do not extend beyond the end of an allocated block. This would be very time-consuming considering how often the heap is used.

The memory used by a heap is private to the process that created it and cannot be shared amongst processes. If, for instance, a DLL creates a private heap, each process that attaches to the DLL will cause a separate instance of the private heap to be created in the virtual address space of the attached client process that is invisible to the other DLL client processes.

Memory Heaps

- A heap is a region of contiguous virtual memory
  - Blocks of memory suballocated from pages at run-time and managed by heap
- Hides details of virtual memory from application
  - page-based granularity
  - reserving and committing pages
- The size of the heap is only limited by the available memory
- One process can have multiple heaps
- Heap handles and memory is private to a process

Every Windows NT process automatically has one heap called the default heap and may optionally nave as many other dynamic heaps as they wish, by creating and destroying them on the fly. There is no difference between them. The Win32 global and local heap memory management functions, like GlobalAlloc() and LocalAlloc(), use the default process heap. The C run-time library memory management functions, like malloc() , may use the default heap, depending on the implementation. The `GetProcessHeap()` function returns a handle to the default heap for a process. This handle can be used in all calls which take a heap handle except HeapDestroy() . Functions such as `GlobalAlloc()` or `malloc()` simply call Get `ProcessHeap()` to retrieve a handle to the default heap, and then manage memory accordingly.

Both default and dynamic heaps have a specific amount of reserved and committed memory regions associated with them initially, but they behave differently with respect to these limits. The default heap's initial reserved and committed memory region sizes are designated when the application is linked. Visual C++ supports the *-HEAPSIZE [reserve] [,commit]* switch to affect the heap or you can link your application with a module definition file and include a *HEAPSIZE [reserve] [commit]* statement in it. By default, commit (4k) is less than reserve (1Mb).

Each application carries this information within its executable image information. You can view this information by dumping header information for the executable image by typing

dumpbin -headers <exename>

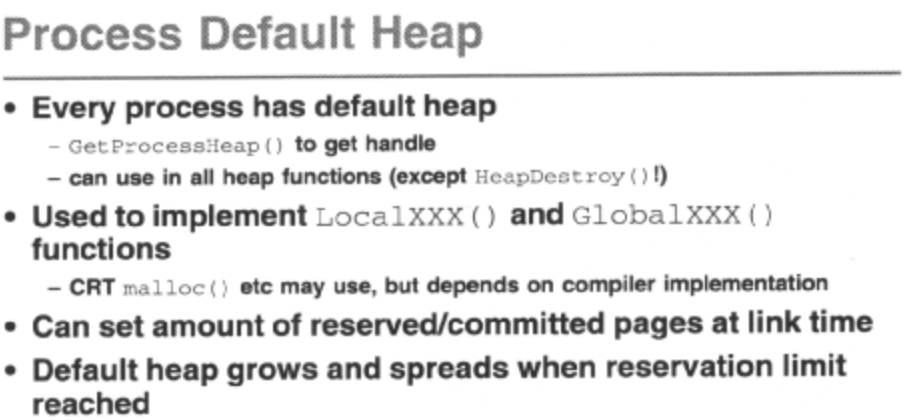
Since the minimum range of address space that can be reserved is 16 pages (64K), the loader will reserve at least 16 pages for the application heap at load time, no matter what the executable image says.

Generally with any heap, and also with the process stack, when commit is less than reserve, memory demands are reduced but execution time is marginally slower. This is because the system sets up guard pages around the heap and stack and could have to process guard page faults. When the heap or stack grows big enough, the guard pages are accessed outside the committed area causing a guard page fault, which tells the system to map in another page. The application continues to run as if you had originally had the new page committed. If the

committed memory is equal to the reserve, no guard pages are created and the program faults if it goes outside the committed memory area.

In some cases the default heap needs to allocate more memory than is available in its current reserved address space. In these cases the default heap may respond differently to a dynamic heap. Instead of failing an allocation request, the default heap manager reserves an additional 1 MB address range elsewhere in the process and commits as much memory as it needs from this reserved address range to satisfy the allocation request. The default heap manager is then responsible for managing this new memory region, transparently to the application, as well as the original heap space. If necessary, it will repeat this as necessary until the process runs out of physical memory and/or logical address space.

That is why managing the initial heap parameters is important. The time taken to locate and reserve a new range of addresses in a process can be saved by reserving a large enough address range initially. Each application has 2 GB of virtual address space and requires relatively little physical memory to support it.



`HeapCreate()` can be used to create multiple private heaps. Since it is possible to have many co-existing dynamic heaps in your process, `HeapCreate()` returns a unique handle to identify the heap, which is used to identify the target heap in other heap functions. Having to identify a heap by it's handle makes managing dynamic heaps more difficult than managing the default heap, since you have to keep each heap handle around for the life of the heap. As mentioned previously, it is possible to get the handle to the default heap using `GetProcessHeap()` and to use the heap functions on this handle. This is very often what you want for small, independent allocations. Similarly you could also use C-runtime memory management functions like malloc(), or local/global heap memory management functions like `GlobalAlloc()` and `LocalAlloc()`.

When creating a heap the maximum size of the heap must be specified, so that the function knows how much address space to reserve, and the amount of memory to be initially committed must also be specified. A process would normally create a heap when it starts up, specifying an initial size sufficient to satisfy the memory requirements of the process. If the `HeapCreate()` call failed, the process could terminate or issue some sort of memory warning; but if it succeeded, the process is assured of having the memory when it needs it.

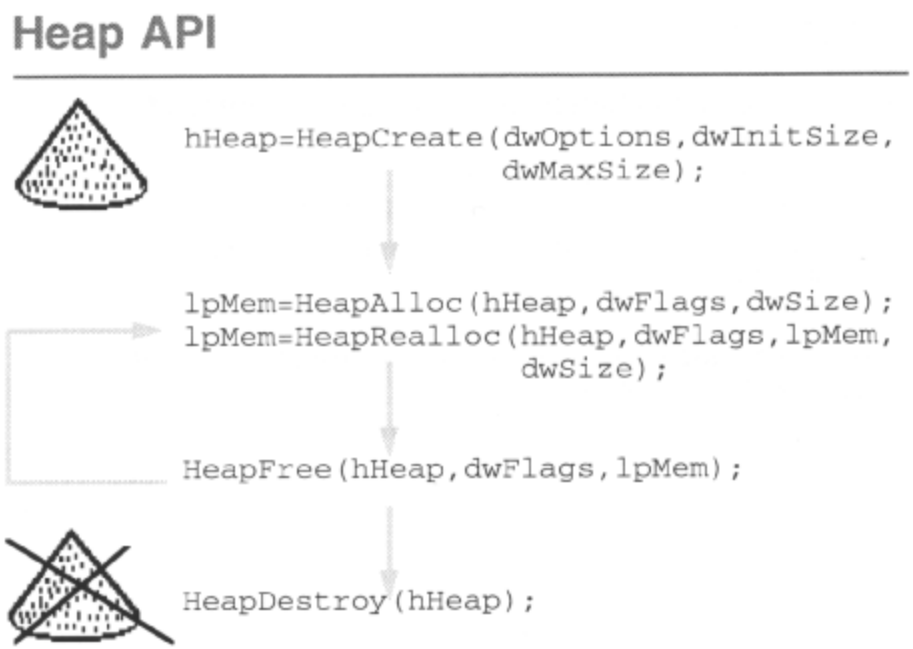
In deterministic cases it is easy to calculate the heap size needed. In other cases, it is more difficult to predict. Dynamic heaps have a provision for this latter circumstance. By specifying a maximum heap size of zero, you make the heap assume a behavior like the default heap discussed already. It will grow and spread in your process as much as necessary, limited only by available memory. In this case, available memory means available virtual address space in the process and available physical memory and pagefile space on the hard disk.

Like all of the heap memory functions, `HeapAlloc()` requires a heap handle as its first argument, and allocates blocks of memory from a private heap and returns a pointer to the allocated block.

`HeapReAlloc()` requires a pointer to a previous allocation in order to re-size it. Although heap memory is not intrinsically "movable" as in global heap and local heap memory, it may be moved during HeapReAlloc() . The pointer returned to the resized block of memory, may not be referencing the same location as initially indicated by the pointer passed to the function. This is the only time memory can be moved in dynamic heaps. It only affects the block of memory referenced in the call to `HeapReAlloc()` and it is application-driven.

A pointer returned by `HeapAlloc()`or `HeapReAlloc()` can be used in `HeapFree()` to release the block, or in `HeapSize()` to determine the size of the block.

A dynamic heap is destroyed by calling HeapDestroy() which invalidates the heap handle. This function will remove a heap regardless of its state and doesn't care whether you have outstanding allocations in the heap or not.



By default, access to each heap created is serialized, which prevents multiple threads in the same process from simultaneously accessing the heap. There is a small performance cost to serialization, but mutual exclusion must be used whenever two threads heap usage might collide with a strong possibility of causing corruption of the heap.

Setting the HEAP_NO_SERIALIZE flag when the heap is created, or when a particular heap function is used, eliminates this default mutual exclusion on the heap. This flag can be safely set if only one thread accesses the heap or if the process uses another form of mutual exclusion, e.g. a mutex object, to keep multiple threads from accessing the heap at the same time. Specifying the HEAP_NO_SERIALIZE flag improves overall heap performance slightly and means the heap needs less memory overhead to maintain itself.

Interestingly enough, access to the default heap is serialized so there is no need to worry about multiple threads in a process corrupting the default heap.

Specifying the HEAP_GENERATE_EXCEPTIONS flag when the heap is created or when memory is allocated or reallocated from a heap, specifies that the heap manager will raise an exception to indicate a heap allocation function failure, rather than returning NIJLL. This flag is a useful feature if you have exception handling built into your application. Exception handling can be an effective way of triggering special events, such as low memory situations, in your application. Exception values on such exceptions are STATUS_NO_MEMORY if the allocation failed for lack of memory or
STATUS_ACCESS_VIOLATION if the allocation failed because of heap corruption or improper function parameters.
Specifying the HEAP_ZERO_MEMORY flag when memory is allocated or reallocated from a heap, specifies that the allocated memory will be initialized to zero values which is not the default.

Memory allocated from a heap is not moveable, so allocating and freeing many small different-size blocks may fragment the heap, over time. However, if a block is reallocated and there is not enough room to reallocate the memory in place, the system will attempt to move the memory block. This will happen by default, unless the HEAP_REALLOC_IN_PLACE_ONLY flag is specified when heap memory is reallocated, in which case the function returns with failure status rather than move the memory. Relocatable blocks will help prevent the heap from becoming fragmented.

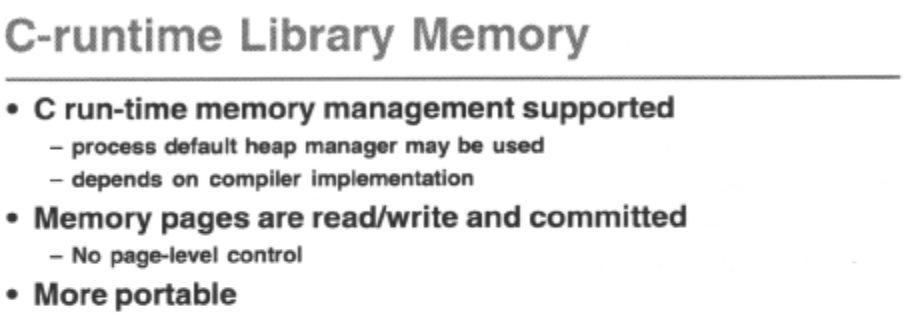
Historically, managing memory in Windows 3.x using the C run-time **(CRT)** library was fraught with difficulty. In Win32 the C-runtime memory allocation routines, such as `malloc(),` are implemented in `the same kind` of way as FIXED memory allocated with `GlobalAlloc()` and `LocalAlloc()`

Whether the default heap manager is used to implement CRT memory functions depends on the C compiler being used.

The C-runtime library with the Windows NT SDK uses the heap manager. In this case the heap manager does not divide its space among the C-runtime functions and the global and local heap memory functions, but treats them the same, promoting consistent behavior.

The C-runtime library with the Visual C++ tool for Windows NT, adopts it own heap management strategy.

As with `GlobalAlloc()` and `LocalAlloc()`, memory pages used are committed, with read/write/execute access.



There are a number of good reasons to use heaps. As mentioned, heap fragmentation may occur over time if many small variable-size memory blocks are sub-allocated and freed. This is because once a block becomes free it may be the wrong size to re-allocate. Using multiple dynamic heaps, which store objects of the same size within them, may be more efficient use of memory. This way freeing one object guarantees another one can fit in it's place.

Also mentioned previously, heap memory blocks are contiguous in the process address space and belong to pages with the same access protection. Overwriting one block when writing to a different block is not considered to be a program error.. The heap manager doesn't make sure that writes do not extend beyond the end of an allocated block. Localizing data objects, as mentioned above, may well also make it easier to spot program bugs that are caused through heap corruption.

A nice application of using multiple heaps to localize the storage of objects could be implemented simply using C++. A C++ class could have overloaded `new()` and `delete()` operators which create a separate heap and perform all object storage allocation from this single heap.
Performance and fragmentation issues are closely linked. If the virtual memory APIs is used to manage memory objects that are used in close proximity to each other in the code, then apart from being wasteful (page-level fragmentation) it is potentially slower. By managing such data objects on the heap, they are likely to co-exist in less pages and it is therefore less likely that the system needs to page as often.

Of course, any heap does have the disadvantage of not allowing page-level control. This means that it is not possible to mark data items on the heap with different access control. Also because pages of committed memory on a heap are not decommitting when finished with, heaps can gobble up both process and system resources. With all types of heap memory, an overhead is associated with the creation of the heap and with each memory allocation from the heap. This overhead is due to the granularity on memory allocations within the heap, the overhead necessary to manage each memory block within the heap and the general heap issues like serialization.

The granularity of heap allocations in Win32 is 8 bytes and the overhead is 8 bytes. So if you request heap memory allocations of 1 byte, the heap returns pointers to blocks of memory, which are 16 byte, aligned. If you request heap memory allocations of 8 bytes, the heap returns pointers to blocks of memory, which are 32 byte, aligned.

If you request heap memory allocations of 9 bytes, the heap returns pointers to blocks of memory which are 24 byte aligned. This is the same for the default heap as for any other dynamic heap. Therefore `HeapAlloc()`, `LocalAlloc()` and `GlobalAlloc()` all work the same. `HeapSize()`, `LocalSize()` and `GlobalSize()` always returns you the size of the block you asked for, not the size allocated. The extra cost of allocating a block of MOVEABLE/DISCARDABLE heap memory with `LocalAlloc()` or `GlobalAlloc()` is 8 bytes in the MOVEABLE memory handle table, although a new committed page is only required every 4096/8 allocations.

As expected the C run-time functions exhibit the same heap memory behavior if they are layered on the default heap. So this is a considerable amount of overhead on smaller allocations. Heap creation overhead varies according to whether serialization is being performed (1432 bytes) or not (1376 bytes).

C-runtime Library Memory

- C run-time memory management supported
  - process default heap manager may be used
  - depends on compiler implementation
- Memory pages are read/write and committed
  - No page-level control
- More portable