times larger than 2GB of physical memory, and there may be many such processes running concurrently.

- The OS maps virtual addresses to physical addresses.
- Most virtual pages will not be in physical memory, so the OS responds to *page faults* (references to pages not in memory) and loads the data from disk, either from the system swap file or from a normal file. Page faults, while transparent to the programmer, have a significant impact on performance, and programs should be designed to minimize faults. Again, many OS texts treat this important subject, which is beyond the scope of this book.

Figure 5–1 shows the Windows memory management API layered on the Virtual Memory Manager. The Virtual Memory Windows API (`VirtualAlloc`, `VirtualFree`, `VirtualLock`, `VirtualUnlock`, and so on) deals with whole pages. The Windows Heap API manages memory in user-defined units.

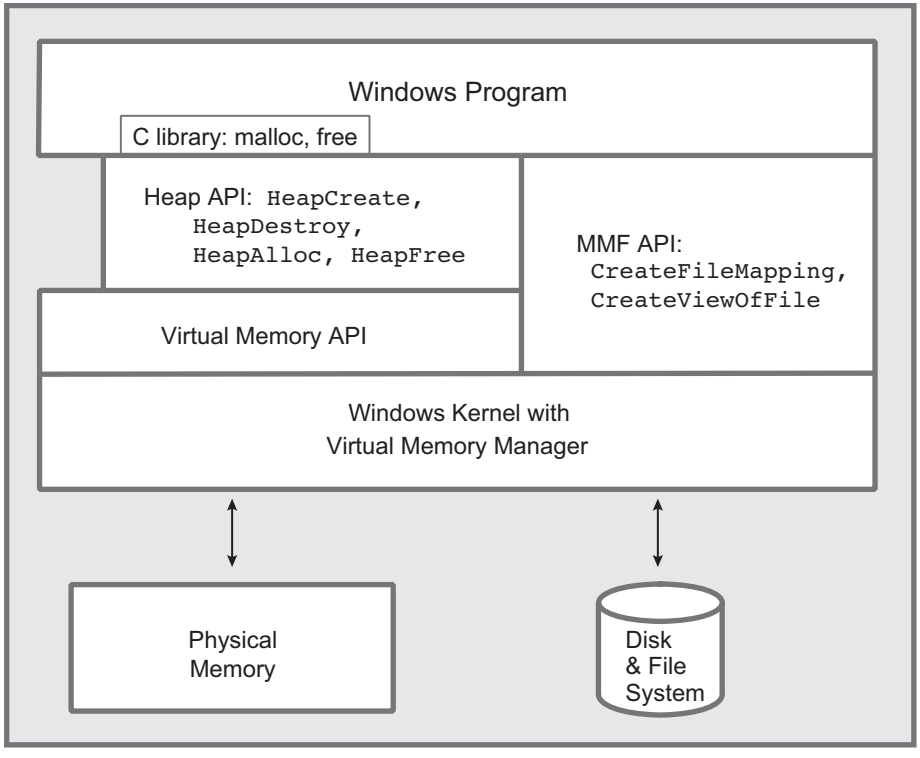

Figure 5–1 Windows Memory Management Architecture

The layout of the virtual memory address space is not shown because it is not directly relevant to the API, and the layout could change in the future. The Microsoft documentation provides this information.

Nonetheless, many programmers want to know more about their environment. To start to explore the memory structure, invoke the following.

```
VOID GetSystemInfo (LPSYSTEM_INFO lpSystemInfo)
```

The parameter is the address of a `PSYSTEM_INFO` structure containing information on the system's page size, allocation granularity, and the application's physical memory address. You can run the `version` program in the *Examples* file to see the results on your computer, and an exercise (with a screenshot) suggests an enhancement.

Heaps

Windows maintains pools of memory in *heaps*. A process can contain several heaps, and you allocate memory from these heaps.

One heap is often sufficient, but there are good reasons, explained below, for multiple heaps. If a single heap is sufficient, just use the C library memory management functions (`malloc`, `free`, `calloc`, `realloc`).

Heaps are Windows objects; therefore, they have handles. However, heaps are not kernel objects. The heap handle is necessary when you're allocating memory. Each process has its own default heap, and the next function obtains its handle.

```
HANDLE GetProcessHeap (VOID)

Return: The handle for the process's heap; NULL on failure.
```

Notice that `NULL` is the return value to indicate failure rather than `INVALID_HANDLE_VALUE`, which is returned by `CreateFile`.

A program can also create distinct heaps. It is convenient at times to have separate heaps for allocation of separate data structures. The benefits of separate heaps include the following.

- **Fairness**. If threads allocate memory solely from a unique heap assigned to the thread, then no single thread can obtain more memory than is allocated to its heap. In particular, a memory leak defect, caused by a program neglecting to free data elements that are no longer needed, will affect only one thread of a process.[3]
- **Multithreaded performance**. By giving each thread its own heap, contention between threads is reduced, which can substantially improve performance. See Chapter 9.
- **Allocation efficiency**. Allocation of fixed-size data elements within a small heap can be more efficient than allocating elements of many different sizes in a single large heap. Fragmentation is also reduced. Furthermore, giving each thread a unique heap for storage used only within the thread simplifies synchronization, resulting in additional efficiencies.
- **Deallocation efficiency**. An entire heap and all the data structures it contains can be freed with a single function call. This call will also free any leaked memory allocations in the heap.
- **Locality of reference efficiency**. Maintaining a data structure in a small heap ensures that the elements will be confined to a relatively small number of pages, potentially reducing page faults as the data structure elements are processed.

The value of these advantages varies depending on the application, and many programmers use only the process heap and the C library. Such a choice, however, prevents the program from exploiting the exception generating capability of the Windows memory management functions (described along with the functions). In any case, the next two functions create and destroy heaps.[4]

Creating a Heap

Use `HeapCreate` to create a new heap, specifying the initial heap size.

The initial heap size, which can be zero and is always rounded up to a multiple of the page size, determines how much physical storage (in a *paging file*) is *committed* to the heap (that is, the required space is allocated from the heap) initially, rather than on demand as memory is allocated from the heap. As a program exceeds the initial size, additional pages are committed automatically up to the maximum size. Because the paging file is a limited resource, deferring commitment is a good practice unless it is known ahead of time how large the

[3] Chapter 7 introduces threads.

[4] In general, create objects of type `X` with the `CreateX` system call. `HeapCreate` is an exception to this pattern.

heap will become. dwMaximumSize, if nonzero, determines the heap's maximum size as it expands dynamically. The process heap will also grow dynamically.

```
HANDLE HeapCreate (
   DWORD flOptions,
   SIZE_T dwInitialSize,
   SIZE_T dwMaximumSize)
```

Return: A heap handle, or NULL on failure.

The two size fields are of type SIZE_T rather than DWORD. SIZE_T is defined to be either a 32-bit or 64-bit unsigned integer, depending on compiler flags (_WIN32 and _WIN64). SIZE_T helps to enable source code portability to both Win32 and Win64. SIZE_T variables can span the entire range of a 32- or 64-bit pointers. SSIZE_T is the signed version but is not used here.

flOptions is a combination of three flags.

- HEAP_GENERATE_EXCEPTIONS—With this option, failed allocations generate an exception for SEH processing (see Chapter 4). HeapCreate itself will not cause an exception; rather, functions such as HeapAlloc, which are explained shortly, cause an exception on failure if this flag is set. There is more discussion after the memory management function descriptions.
- HEAP_NO_SERIALIZE—Set this flag under certain circumstances to get a small performance improvement; there is additional discussion after the memory management function descriptions.
- HEAP_CREATE_ENABLE_EXECUTE—This is an out-of-scope advanced feature that allows you to specify that code can be executed from this heap. Normally, if the system has been configured to enforce data execution prevention (DEP), any attempt to execute code in the heap will generate an exception with code STATUS_ACCESS_VIOLATION, partially providing security from code that attempts to exploit buffer overruns.

There are several other important points regarding dwMaximumSize.

- If dwMaximumSize is nonzero, the virtual address space is allocated accordingly, even though it may not be committed in its entirety. This is the maximum size of the heap, which is said to be *nongrowable*. This option limits a heap's size, perhaps to gain the fairness advantage cited previously.

- If, on the other hand, `dwMaximumSize` is `0`, then the heap is *growable* beyond the initial size. The limit is determined by the available virtual address space not currently allocated to other heaps and swap file space.

Note that heaps do not have security attributes because they are not kernel objects; they are memory blocks managed by the heap functions. File mapping objects, described later in the chapter, can be secured (Chapter 15).

To destroy an entire heap, use `HeapDestroy`. `CloseHandle` is not appropriate because heaps are not kernel objects.

```
BOOL HeapDestroy (HANDLE hHeap)
```

`hHeap` should specify a heap generated by `HeapCreate`. Be careful not to destroy the process's heap (the one obtained from `GetProcessHeap`). Destroying a heap frees the virtual memory space and physical storage in the paging file. Naturally, well-designed programs should destroy heaps that are no longer needed.

Destroying a heap is also a quick way to free data structures without traversing them to delete one element at a time, although C++ object instances will not be destroyed inasmuch as their destructors are not called. Heap destruction has three benefits.

1. There is no need to write the data structure traversal code.
2. There is no need to deallocate each individual element.
3. The system does not spend time maintaining the heap since all data structure elements are deallocated with a single call.

The C library uses only a single heap. There is, therefore, nothing similar to Windows heap handles.

The UNIX `sbrk` function can increase a process's address space, but it is not a general-purpose memory manager.

UNIX does not generate signals when memory allocation fails; the programmer must explicitly test the returned pointer.

Managing Heap Memory

The heap management functions allocate and free memory blocks.

HeapAlloc

Obtain memory blocks from a heap by specifying the heap's handle, the block size, and several flags.

```
LPVOID HeapAlloc (
   HANDLE hHeap,
   DWORD dwFlags,
   SIZE_T dwBytes)
```

Return: A pointer to the allocated memory block, or `NULL` on failure (unless exception generation is specified).

Parameters

`hHeap` is the heap handle for the heap in which the memory block is to be allocated. This handle should come from either `GetProcessHeap` or `HeapCreate`.

`dwFlags` is a combination of three flags:

- `HEAP_GENERATE_EXCEPTIONS` and `HEAP_NO_SERIALIZE`—These flags have the same meaning as for `HeapCreate`. The first flag is ignored if it was set with the heap's `HeapCreate` function and enables exceptions for the specific `HeapAlloc` call, even if `HEAP_GENERATE_EXCEPTIONS` was not specified by `HeapCreate`. The second flag should not be used when allocating within the process heap, and there is more information at the end of this section.
- `HEAP_ZERO_MEMORY`—This flag specifies that the allocated memory will be initialized to `0`; otherwise, the memory contents are not specified.

`dwBytes` is the size of the block of memory to allocate. For nongrowable heaps, this is limited to `0x7FFF8` (approximately 0.5MB). This block size limit applies even to Win64 and to very large heaps.

The return value from a successful `HeapAlloc` call is an `LPVOID` pointer, which is either 32 or 64 bits, depending on the build option.

Note: Once `HeapAlloc` returns a pointer, use the pointer in the normal way; there is no need to make reference to its heap.

Heap Management Failure

The `HeapAlloc` has a different failure behavior than other functions we've used.

- Function failure causes an exception when using HEAP_GENERATE_EXCEPTIONS. The exception code is either STATUS_NO_MEMORY or STATUS_ACCESS_VIOLATION.
- Without HEAP_GENERATE_EXCEPTIONS, HeapAlloc returns a NULL pointer.
- In either case, you cannot use GetLastError for error information, and hence you cannot use this book's ReportError function to produce a text error message.

HeapFree

Deallocating memory from a heap is simple.

```
BOOL HeapFree (
   HANDLE hHeap,
   DWORD dwFlags,
   LPVOID lpMem)
```

dwFlags should be 0 or HEAP_NO_SERIALIZE (see the end of the section). lpMem should be a value returned by HeapAlloc or HeapReAlloc (described next), and, of course, hHeap should be the heap from which lpMem was allocated.

A FALSE return value indicates a failure, and you can use GetLastError, which does not work with HeapAlloc. HEAP_GENERATE_EXCEPTIONS does not apply to HeapFree.

HeapReAlloc

Memory blocks can be reallocated to change their size. Allocation failure behavior is the same as with HeapAlloc.

```
LPVOID HeapReAlloc (
   HANDLE hHeap,
   DWORD dwFlags,
   LPVOID lpMem,
   SIZE_T dwBytes)
```

Return: A pointer to the reallocated block. Failure returns NULL or causes an exception.

Parameters

The first parameter, `hHeap`, is the same heap used with the `HeapAlloc` call that returned the `lpMem` value (the third parameter). `dwFlags` specifies some essential control options.

- `HEAP_GENERATE_EXCEPTIONS` and `HEAP_NO_SERIALIZE`—These flags are the same as for `HeapAlloc`.
- `HEAP_ZERO_MEMORY`—Only newly allocated memory (when `dwBytes` is larger than the original block) is initialized. The original block contents are not modified.
- `HEAP_REALLOC_IN_PLACE_ONLY`—This flag specifies that the block cannot be moved. When you're increasing a block's size, the new memory must be allocated at the address immediately after the existing block.

`lpMem` specifies the existing block in `hHeap` to be reallocated.

`dwBytes` is the new block size, which can be larger or smaller than the existing size, but, as with `HeapAlloc`, it must be less than `0x7FFF8`.

It is possible that the returned pointer is the same as `lpMem`. If, on the other hand, a block is moved (permit this by omitting the `HEAP_REALLOC_IN_PLACE_ONLY` flag), the returned value might be different. Be careful to update any references to the block. The data in the block is unchanged, regardless of whether or not it is moved; however, some data will be lost if the block size is reduced.

HeapSize

Determine the size of an allocated block by calling `HeapSize` with the heap handle and block pointer. This function could have been named `HeapGetBlockSize` because it does not obtain the heap size. The value will be greater than or equal to the size used with `HeapAlloc` or `HeapReAlloc`.

```
SIZE_T HeapSize (
   HANDLE hHeap,
   DWORD dwFlags,
   LPCVOID lpMem)
```

Return: The size of the block, or zero on failure.

The only possible `dwFlags` value is `HEAP_NO_SERIALIZE`.

The error return value is `(SIZE_T)-1`. You cannot use `GetLastError` to find extended error information.

More about the Serialization and Exceptions Flags

The heap management functions use two unique flags, `HEAP_NO_SERIALIZE` and `HEAP_GENERATE_EXCEPTIONS`, that need additional explanation.

The `HEAP_NO_SERIALIZE` Flag

The functions `HeapCreate`, `HeapAlloc`, and `HeapReAlloc` can specify the `HEAP_NO_SERIALIZE` flag. There can be a small performance gain with this flag because the functions do not provide mutual exclusion to threads accessing the heap. Some simple tests that do nothing except allocate memory blocks measured a performance improvement of about 16 percent. This flag is safe in a few situations, such as the following.

- The program does not use threads (Chapter 7), or, more accurately, the process (Chapter 6) has only a single thread. All examples in this chapter use the flag.
- Each thread has its own heap or set of heaps, and no other thread accesses those heaps.
- The program has its own mutual exclusion mechanism (Chapter 8) to prevent concurrent access to a heap by several threads using `HeapAlloc` and `HeapReAlloc`.

The `HEAP_GENERATE_EXCEPTIONS` Flag

Forcing exceptions when memory allocation fails avoids the need for error tests after each allocation. Furthermore, the exception or termination handler can clean up memory that did get allocated. This technique is used in some examples.

Two exception codes are possible.

1. `STATUS_NO_MEMORY` indicates that the system could not create a block of the requested size. Causes can include fragmented memory, a nongrowable heap that has reached its limit, or even exhaustion of all memory with growable heaps.
2. `STATUS_ACCESS_VIOLATION` indicates that the specified heap has been corrupted. For example, a program may have written memory beyond the bounds of an allocated block.

Setting Heap Information

`HeapSetInformation` allows you to enable the "low-fragmentation" heap (LFH) on NT5 (Windows XP and Server 2003) computers; the LFH is the default on NT6. This is a simple function; see MSDN for an example. The LFH can help program performance when allocating and deallocating small memory blocks with different sizes.

`HeapSetInformation` also allows you to enable the "terminate on corruption" feature. Windows terminates the process if it detects an error in the heap; such an error could occur, for example, if you wrote past the bounds of an array allocated on the heap.

Use `HeapQueryInformation` to determine if the LFH is enabled for the heap. You can also determine if the heap supports look-aside lists (see MSDN).

Other Heap Functions

`HeapCompact`, despite the name, does not compact the heap. However, it does return the size of the largest committed free block in the heap. `HeapValidate` attempts to detect heap corruption. `HeapWalk` enumerates the blocks in a heap, and `GetProcessHeaps` obtains all the heap handles that are valid in a process.

`HeapLock` and `HeapUnlock` allow a thread to serialize heap access, as described in Chapter 8.

Some functions, such as `GlobalAlloc` and `LocalAlloc`, were used for compatibility with 16-bit systems and for functions inherited from 16-bit Windows. These functions are mentioned first as a reminder that some functions continue to be supported even though they are not always relevant and you should use the heap functions. However, there are cases where MSDN states that you need to use these functions, and memory must be freed with the function corresponding to its allocator. For instance, use `LocalFree` with `FormatMessage` (see Program 2–1, `ReportError`). In general, if a function allocates memory, read MSDN to determine the correct free function, although `FormatMessage` is the only such function used in this book.

Summary: Heap Management

The normal process for using heaps is straightforward.

1. Get a heap handle with either `CreateHeap` or `GetProcessHeap`.
2. Allocate blocks within the heap using `HeapAlloc`.
3. Optionally, free some or all of the individual blocks with `HeapFree`.

4. Destroy the heap and close the handle with `HeapDestroy`.
5. The C run-time library (`malloc`, `free`, etc.) are frequently adequate. However, memory allocated with the C library must be freed with the C library. You cannot assume that the C library uses the process heap.

Figure 5–2 and Program 5–1 illustrate this process.

Normally, programmers use the C library `<stdlib.h>` memory management functions and can continue to do so if separate heaps or exception generation are not needed. `malloc` is then logically equivalent to `HeapAlloc` using the process heap, `realloc` to `HeapReAlloc`, and `free` to `HeapFree`. `calloc` allocates and initializes objects, and `HeapAlloc` can easily emulate this behavior. There is no C library equivalent to `HeapSize`.

Example: Sorting Files with a Binary Search Tree

A search tree is a common dynamic data structure requiring memory management. Search trees are a convenient way to maintain collections of records, and they have the additional advantage of allowing efficient sequential traversal.

Program 5–1 implements a sort (`sortBT`, a limited version of the UNIX `sort` command) by creating a binary search tree using two heaps. The keys go into the *node heap*, which represents the search tree. Each node contains left and right pointers, a key, and a pointer to the data record in the *data heap*. The complete record, a line of text from the input file, goes into the data heap. Notice that the node heap consists of fixed-size blocks, whereas the data heap contains strings with different lengths. Finally, tree traversal displays the sorted file.

This example arbitrarily uses the first 8 bytes of a string as the key rather than using the complete string. Two other sort implementations in this chapter (Programs 5–4 and 5–5) sort files.

Figure 5–2 shows the sequence of operations for creating heaps and allocating blocks. The program code on the right is *pseudocode* in that only the essential function calls and arguments are shown. The virtual address space on the left shows the three heaps along with some allocated blocks in each. The figure differs slightly from the program in that the root of the tree is allocated in the process heap in the figure but not in Program 5–1. Finally, the figure is not drawn to scale.

Note: The actual locations of the heaps and the blocks within the heaps depend on the Windows implementation and on the process's history of previous memory use, including heap expansion beyond the original size. Furthermore, a growable heap may not occupy contiguous address space after it grows beyond the originally committed size. The best programming practice is to make no assumptions; just use the memory management functions as specified.

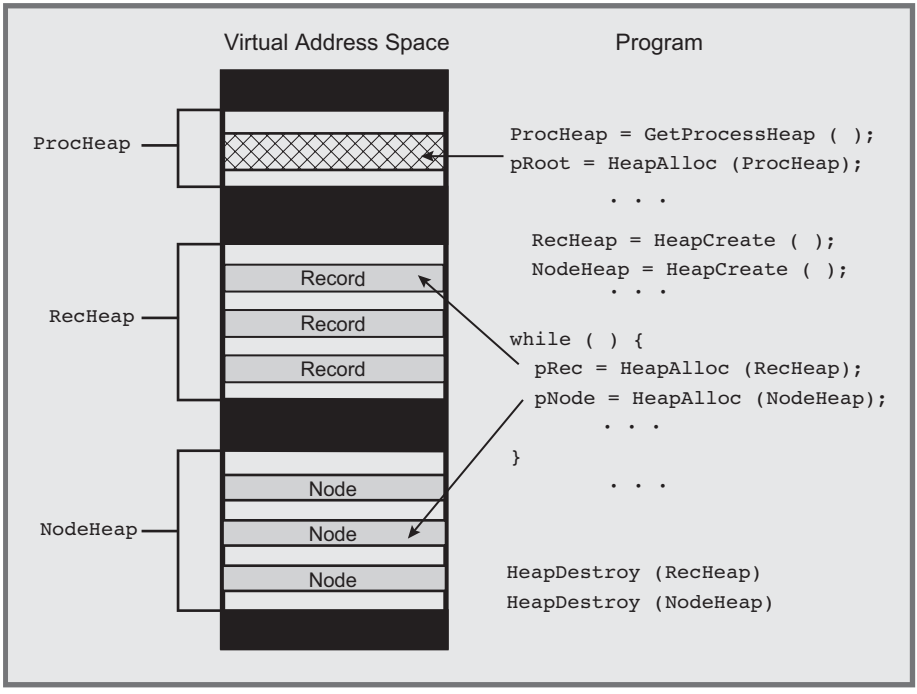

Figure 5–2 Memory Management in Multiple Heaps

Program 5–1 illustrates some techniques that simplify the program and would not be possible with the C library alone or with the process heap.

- The node elements are fixed size and go in a heap of their own, whereas the varying-length data element records are in a separate heap.
- The program prepares to sort the next file by destroying the two heaps rather than freeing individual elements.
- Allocation errors are processed as exceptions so that it is not necessary to test for `NULL` pointers.

An implementation such as Program 5–1 is limited to smaller files when using Windows because the complete file and a copy of the keys must reside in virtual memory. The absolute upper limit of the file length is determined by the available virtual address space (3GB at most for Win32; the practical limit is less). With Win64, you will probably not hit a practical limit.