

The Virtual...() Functions

At the lowest level of Win32's memory management functions come those that work directly with virtual memory. We'll mention them here, although most of the normal things you want to do in an application program can usually be handled using either language-specific constructs or the `Global...()` functions.

One time when you may want to work with memory at this level is when you *might* need a large contiguous area of memory, but won't be sure until run-time. These calls let you reserve a range of memory addresses for future use, and you can then allocate memory to use those addresses, in the sure knowledge that they won't be allocated to anyone else.

Let's start with `VirtualAlloc()` which allows you to reserve or commit an area of memory within the address space of the calling process. You can use the function to perform the following operations:

- ▶ Reserve a region of free pages
- ▶ Commit some or all of the pages reserved by a previous call to `VirtualAlloc()`
- ▶ Reserve and commit a region of pages in one go

A **reserved** page is one which cannot be used by other memory allocation functions, but which has no physical storage associated with it, and so is not accessible. Before a reserved page can be used it must be **committed**, when storage is allocated for the page and access protection applied. The system only initializes and loads a committed page at the first read or write operation. It is not an error to try to commit a page which is already committed — the request will work, but will do nothing.

This means that you can use the function to reserve a block of pages within the process's virtual address space, an operation which doesn't consume any physical storage but which guarantees that a certain range of virtual memory will be available as required. Then, as pages are actually needed, they can be committed, and it is only at this point that they'll take up any physical memory.

Creating And Destroying Pages

Here's what the function looks like:

```
LPVOID VirtualAlloc(
    LPVOID pAddress,           // address of start of region
    DWORD dwSize,             // size of region in bytes
    DWORD dwAllocType,        // type of allocation
    DWORD dwProtect);        // access protection
```

The first two parameters are just what you'd expect — the address at which you want allocation to start and the size of the region you want to allocate. If you aren't concerned where the region is allocated, you can pass `NULL` for the first parameter, and the system will place it for you.

If successful, the function returns the starting address of the allocated region, which might not be the same as the address you specified. This is because when reserving pages the address is rounded down to the nearest 64KB boundary, and when committing it is rounded down to the nearest page boundary.

The size you give determines how many pages are reserved or committed. Note that even if only a single byte falls in a new page, that entire page must be included in the allocation.

The third parameter in the call to `VirtualAlloc()` specifies the type of the allocation, and can be any combination of the following flags:

- ▶ `MEM_COMMIT`, which allocates physical storage in memory or the paging file for the range of pages specified
- ▶ `MEM_RESERVE`, which reserves a range of the process's virtual address space, without allocating any actual storage. Reserved pages can't be used with any other allocation function until they have been released
- ▶ `MEM_TOP_DOWN`, which allocates memory at the highest possible address

The final parameter specifies the access protection given to the page, and can be any one of the following flags:

- ▶ `PAGE_READONLY`, which allows only read access to the pages
- ▶ `PAGE_READWRITE`, which allows read and write access
- ▶ `PAGE_EXECUTE`, which allows execute access only (any attempt to read or write will cause an access violation)
- ▶ `PAGE_EXECUTE_READ`, which allows execute and read access
- ▶ `PAGE_EXECUTE_READWRITE`, which allows execute, read and write access
- ▶ `PAGE_NOACCESS`, which disables all access to the page (any attempt to access the page results in a General Protection Fault)

Note that some of these access types may not be supported on all Win32 platforms.

Two modifiers can be applied to all access protection flags apart from `PAGE_NOACCESS`:

- ▶ `PAGE_GUARD`, which turns the pages in the region into guard pages (see below)
- ▶ `PAGE_NOCACHE`, which does not permit caching of the pages in the region

Any attempt to read or write a page marked as a guard page will cause the operating system to throw a `STATUS_GUARD_PAGE` exception, and they thus act as an access alarm. They are often used to place guards around data structures, so that it is possible to detect any attempt to write outside the boundaries of the structure.

Note that once a guard page has been 'triggered', the system removes the `PAGE_GUARD` status, and the page reverts to its underlying access protection type.

Once you've finished with a block of memory, you can release it using `VirtualFree()`:

```

BOOL VirtualFree(
    LPVOID pAddress,           // address of region to free
    DWORD dwSize,            // size of region to free
    DWORD dwType);          // type of free operation

```

As with `VirtualAlloc()` you can use this function to do more than one thing. You can:

- ▶ Decommit a region of committed pages
- ▶ Release a region of reserved pages
- ▶ Decommit and release a region of committed pages

The first two parameters specify the address of the block and the size of the region you want to free. If you're decommitting memory, the size will determine how many pages are decommitted. If you're releasing memory, the size must be zero or the function will fail.

The third parameter determines what happens, and must be one of:

- ▶ `MEM_DECOMMIT` to decommit pages
- ▶ `MEM_RELEASE` to release pages

If you're going to release a region of pages, they must all be in the same state (committed or reserved) and you must release the entire region at once. If some of the pages are committed and some reserved, you must first decommit the committed pages with a call to `VirtualFree()`, and then call it again to release the range. If the pages aren't compatible with the operation you're trying to perform on them, the free operation will fail and no pages will be freed.

We should also note the `VirtualAllocEx()` function, which is identical to `VirtualAlloc()`, except that it has an extra parameter which allows you to specify the handle of another process in which to allocate memory. There's also a matching `VirtualFreeEx()` function, and you need to have `PROCESS_VM_OPERATION` access to the process if you want to use these functions.

Other Virtual Memory Functions

You can set or change the protection of pages using `VirtualProtect()`:

```

BOOL VirtualProtect(
    LPVOID pAddress,           // address of region
    DWORD dwSize,             // size of region
    DWORD dwProt,             // new protection to apply
    PDWORD pdwOldProt);      // old protection returned

```

You can pass any of the access protection flags used with `VirtualAlloc()`. If you pass `NULL` for the last parameter, the previous protection won't be returned.

`VirtualLock()` can be used to lock a page or range of pages in memory so that subsequent access doesn't generate a page fault:

```

BOOL VirtualLock(
    LPVOID pAddress,           // address of region to lock
    DWORD dwSize);           // size of region in bytes

```

All pages in a region must be committed before they can be locked, and you can't lock pages which have the `PAGE_NOACCESS` flag set. The number of pages that a process can lock is

limited to 30 by default. This low value reflects the fact that locking pages in memory can have a severe impact on system performance, and you should never lock pages unless it is strictly necessary.

Once you've finished with these pages, you should call `VirtualUnlock()` to remove the lock:

```

BOOL VirtualUnlock(
    LPVOID pAddress,           // address of region to lock
    DWORD dwSize);           // size of region in bytes

```

Note that unlike `GlobalLock()` there is no lock count associated with virtual pages, so it isn't necessary to call `VirtualUnlock()` more than once.

Locking virtual pages isn't implemented on Windows 95/98, so `VirtualLock()` is implemented as a stub function which always returns `TRUE`.

`VirtualQuery()` returns information about a range of pages:

```

DWORD VirtualQuery(
    LPCVOID pAddress,         // address of region
    PMEMORY_BASIC_INFORMATION pBuff, // address of information buffer
    DWORD dwLength);        // size of buffer

```

The `MEMORY_BASIC_INFORMATION` structure gives you a variety of information about a block of memory:

```

typedef struct _MEMORY_BASIC_INFORMATION
{
    PVOID BaseAddress;        // base address of region
    PVOID AllocationBase;    // allocation base address
    DWORD AllocationProtect; // initial access protection
    DWORD RegionSize;        // size, in bytes, of region
    DWORD State;             // committed, reserved, free
    DWORD Protect;           // current access protection
    DWORD Type;              // type of pages
} MEMORY_BASIC_INFORMATION;

```

The `BaseAddress` member tells you the base address of the range of pages within which the address that you passed to `VirtualQuery()` falls. `AllocationBase` is the base address of the original range of pages allocated by `VirtualAlloc()`. The access protection applied when the region was originally allocated (such as `PAGE_READONLY`, `PAGE_READWRITE` and so on) is given by `AllocationProtect`, while `Protect` specifies the current access flag for the region. `RegionSize` tells you the size of the region, beginning at the base, within which all pages have the same attributes. The state of the pages in the region is given by `State`, which can take one of the following values:

- ▶ `MEM_COMMIT`, denotes committed pages for which physical memory has been allocated
- ▶ `MEM_FREE`, denotes free pages available for allocation
- ▶ `MEM_RESERVE`, denotes reserved pages — virtual memory reserved but no physical memory allocated

Finally, `Type` tells you whether the pages in the region are private (`MEM_PRIVATE`), are mapped onto the view of a section (`MEM_MAPPED`) or mapped onto the view of an image section (`MEM_IMAGE`).

The return value from the function tells you how many bytes were returned in the buffer.

`VirtualQueryEx()` does the same thing as `VirtualQuery()`, but also allows you to query memory blocks in another process.

Try It Out — Using Virtual Memory

The following sample program shows how to allocate, and query, virtual memory using the functions we've described:

```
// VirtualMemory.cpp

#include <windows.h>#include <iostream>using namespace std;

void printMemInfo(MEMORY_BASIC_INFORMATION& mbi)
{
    // Size
    cout << "Region size = " << mbi.RegionSize << endl;

    // Allocation protection
    cout << "Allocation Protection: ";
    if (mbi.AllocationProtect & PAGE_READONLY) cout << "Read-only ";
    if (mbi.AllocationProtect & PAGE_READWRITE) cout << "Read-write ";
    if (mbi.AllocationProtect & PAGE_EXECUTE) cout << "Execute-only ";
    if (mbi.AllocationProtect & PAGE_EXECUTE_READ) cout << "Execute-read ";
    if (mbi.AllocationProtect & PAGE_EXECUTE_READWRITE) cout
        << "Execute-read-write ";
    if (mbi.AllocationProtect & PAGE_NOACCESS) cout << "No access ";
    cout << endl;

    // Protection
    cout << "Protection: ";
    if (mbi.Protect & PAGE_READONLY) cout << "Read-only ";
    if (mbi.Protect & PAGE_READWRITE) cout << "Read-write ";
    if (mbi.Protect & PAGE_EXECUTE) cout << "Execute-only ";
    if (mbi.Protect & PAGE_EXECUTE_READ) cout << "Execute-read ";
    if (mbi.Protect & PAGE_EXECUTE_READWRITE) cout << "Execute-read-write ";
    if (mbi.Protect & PAGE_NOACCESS) cout << "No access ";
    cout << endl;

    // State
    cout << "State: ";
    if (mbi.State & MEM_COMMIT) cout << "Committed ";
    if (mbi.State & MEM_RESERVE) cout << "Reserved ";
    if (mbi.State & MEM_FREE) cout << "Free ";
    cout << endl;
}
```

```

// Type
cout << "Type: ";
if (mbi.Type & MEM_PRIVATE) cout << "Private ";
if (mbi.Type & MEM_MAPPED) cout << "Mapped ";
if (mbi.Type & MEM_IMAGE) cout << "Image ";
cout << endl;
}

```

The first step is to provide a function to write out the interesting parts of a `MEMORY_BASIC_INFORMATION` structure. This function takes one of these structures and starts by simply finding the size of the region in question. The various flags that are set for the `AllocationProtect` member are found by taking logical combinations of these flags with `AllocationProtect`, and the results are output to the screen. This is then repeated for the `Protect`, `State` and `Type` members of the structure.

Now let's look at the main body of the code:

```

int main()
{
    int* pInt, *pInt2;

    // Reserve a block of memory
    pInt = static_cast<int*>( VirtualAlloc(0,           // let the system find the
                               // address
                               20000 * sizeof(int), // grab a lot of bytes
                               MEM_RESERVE,         // only reserve it here
                               PAGE_READWRITE))    // read-write access

    if (!pInt)
    {
        cout << "Error reserving pages (" << GetLastError() << ")" << endl;
        return -1;
    } // See what we've got
        MEMORY_BASIC_INFORMATION mbi;
        DWORD dwSize = VirtualQuery(pInt, &mbi, sizeof(mbi));
        cout << "Query reserved block:" << endl;

    printMemInfo(mbi);

    // Now commit a page
    pInt2 = static_cast<int*>( VirtualAlloc(pInt,      // let the system find the
                               // address
                               500 * sizeof(int),    // grab a page
                               MEM_COMMIT,          // commit the memory
                               PAGE_READWRITE));    // read-write access

    if (!pInt2)
    {
        cout << "Error committing pages (" << GetLastError() << ")" << endl;
        return -1;
    } dwSize = VirtualQuery(pInt2, &mbi, sizeof(mbi));
        cout << endl << "Query committed block:" << endl;

    printMemInfo(mbi);

    // Decommit the committed pages
    if (!VirtualFree(pInt2, 500 * sizeof(int), MEM_DECOMMIT))
        cout << "Freeing committed pages failed (" << GetLastError() << ")" << endl;
    else

```

```

    cout << "Pages decommitted" << endl;

    // Release all pages
    if (!VirtualFree(pInt, 0, MEM_RELEASE))
        cout << "Freeing reserved pages failed (" << GetLastError() << ")" << endl;
    else
        cout << "Pages released" << endl;

    return 0;
}

```

If you try building and running this code, you should see output something like this:

```

D:\WINNT\System32\cmd.exe
D:\BegNT\VirtualMemory\Debug>virtualmemory
Query reserved block:
Region size = 81920
Allocation Protection: Read-write
Protection:
State: Reserved
Type: Private

Query committed block:
Region size = 4096
Allocation Protection: Read-write
Protection: Read-write
State: Committed
Type: Private
Pages decommitted
Pages released

D:\BegNT\VirtualMemory\Debug>

```

Let's see what's going on here. The main routine begins by reserving a block of memory:

```

// Reserve a block of memory
pInt = static_cast<int*>( VirtualAlloc(0,           // let the system find the
                          // address
                          20000 * sizeof(int),    // grab a lot of bytes
                          MEM_RESERVE,           // only reserve it here
                          PAGE_READWRITE));      // read-write access

```

In our call to `VirtualAlloc()`, we reserve enough space for 20000 integers, giving it read-write access, and we let the system allocate the address. Once we've done this, we query the block for its attributes using `VirtualQuery()`, and then output the information from the resulting `MEMORY_BASIC_INFORMATION` structure using our `printMemInfo()` function.

Notice that the region size of the reserved block is bigger than the 80000 bytes we asked for, because it is rounded up to the next whole page. The allocation protection shows what we asked for when we called `VirtualAlloc()`, but there is no protection yet because this memory hasn't been committed. The state of the block is reserved, and it is private to our process.

Next, we commit a page from the range we've reserved:

```
// Now commit a page
pInt2 = static_cast<int*>( VirtualAlloc(pInt,      // let the system find the
                           // address
                           500 * sizeof(int),    // grab a page
                           MEM_COMMIT,         // commit the memory
                           PAGE_READWRITE));    // read-write access
```

Again, we query the block and print out the information.

The information tells us that we have a block of 4096 bytes, which is committed. Again, this figure is rounded up to the nearest page. We can also see that the protection has been set to read-write.

The final stage in the process is to decommit and release the pages, which we have to do in two stages because not all of the original allocation is in the same state.

Summary

In this chapter we've looked at the use of storage in Win32, and we started off by looking at files.

We've seen how to use `CreateFile()` to open or create files, and seen that it can also be used for other objects, besides files, as well. The basic operations are `ReadFile()` and `WriteFile()`, which let us read and write at the byte level, and the 'Ex' versions have the important ability to use asynchronous I/O, where we let NT go away and do the operation for us, whilst we get on with other tasks. Win32 also provides a plethora of other file manipulation functions, including the ability to be notified of changes to files and directories.

The second part of the chapter was concerned with using memory, and we saw how there are three ways to use memory under NT — using language-specific features, such as 'new' or 'malloc', using the higher-level Win32 methods, such as `GlobalAlloc()`, and working directly with virtual memory using the `Virtual...`() functions. We pointed out that you should use the highest level approach possible, and that you'll fairly seldom need to use virtual memory directly.