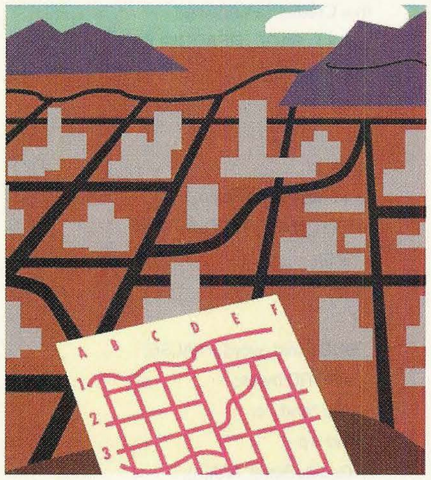
By Douglas A. Hamilton

Navigate Your Way with Mapped Files



SCOTT BALDWIN

ONE OF Windows NT's interesting features is mapped files. Basically, the idea is that with NT's large, flat, virtual address space, there's no reason for an application to have to fool around with tedious I/O operations to read or write a file.

Instead, you use the virtual memory paging mechanism to map the whole file into the address space of the application. Accessing individual characters in the file amounts to dereferencing a pointer. If the address corresponds to a location in the file that has not already been physically read in from the disk, the virtual memory system traps the page fault, reading in the desired data.

What's nice about this mechanism is that it makes it unnecessary for an application to deal with questions of buffering, how big the buffers should be or data that happens to fall across buffer boundaries. It's especially convenient if the data has to be accessed more or less randomly: Doing that with ordinary I/O operations can be a real mess, but it's trivial with mapped files.

The other big advantage is that mapped files are fast. In some simple experiments using the code that I'll show you here—reading files and counting the number of words in them—mapped files were as much as 36% faster than using the NT kernel's ReadFile primitives.

Let's start with a simple example, wc1.c, using ordinary ReadFile calls:

```
#include <windows.h>
#include <stdio.h>
void main( int argc, char **argv )
{
int total = 0, i;
for (i = 1; i < argc; i++)
{
HANDLE f;
int words = 0;
f = CreateFile(argv[i],
  GENERIC_READ,
  FILE_SHARE_READ |
   FILE_SHARE_WRITE,
  NULL, OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL,
  NULL);
if (f != INVALID_HANDLE_VALUE)
{
BOOL mid_word = FALSE;
DWORD bytes;
char buffer[10240], *c;
while (ReadFile(f, buffer,
  sizeof(buffer), &bytes,
  NULL) && bytes)
 for (c = buffer;
  c < buffer + bytes;
  c++)
  switch (*c)
  {
  case '\n':
  case ' ':
  case '\t':
        case ‘\r’:
          if (mid_word)
            {
            mid_word = FALSE;
            words++;
            }
          break;
        default:
          mid_word = TRUE;
        }
    CloseHandle(f);
    printf(“%6d %s\n”, words,
     argv[i]);
    total += words;
    }
  }
if (argc > 2)
 printf(“%6d Total\n”, total);
fflush(stdout);
ExitProcess(0);
}
```

For each of the filenames specified on the command line, this program will open the file and then, reading 10KB chunks of characters at a time, look for word breaks consisting of white-space characters. After a file has been read, the total number of words found will be printed. At the end, if there is more than one file, a total for all the files will be printed. Here's some sample output:

```
121 wc1.c
146 wc2.c
267 Total
```

(The performance of this routine does depend somewhat on the size of the buffer you use. If all the files it's reading are small, it won't make any difference. But, for example, if you shrink the buffer down to only 2KB with a series of files averaging around 46KB, runtime will increase by around 20%. Increasing buffer size beyond 10KB has diminishing returns, so my comparisons are against that buffer size.)

Now here's that same function, recast as wc2.c using mapped files:

```
#include <windows.h>
#include <stdio.h>
void main( int argc, char **argv )
 {
 int total = 0, i;
 for (i = 1; i < argc; i++)
  {
  HANDLE f;
  int words = 0;
  f = CreateFile(argv[i],
    GENERIC_READ,
    FILE_SHARE_READ |
     FILE_SHARE_WRITE,
    NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
  if (f != INVALID_HANDLE_VALUE)
   {
   HANDLE m;
   m = CreateFileMapping(f,
     NULL, PAGE_READONLY,
     0, 0, NULL);
   if (m)
    {
    char *p;
    p = MapViewOfFile(m,
      FILE_MAP_READ, 0,
      0, 0);
    if (p)
     {
     BOOL mid_word = FALSE;
     DWORD bytes;
     char *end, *c;
     end = p +
      GetFileSize(f, NULL);
     for (c = p; c < end; c++)
      switch (*c)
       {
       case ‘\n’:
       case ‘ ’:
       case ‘\t’:
       case ‘\r’:
        if (mid_word)
         {
         mid_word = FALSE;
         words++;
         }
        break;
       default:
        mid_word = TRUE;
       }
     UnmapViewOfFile(p);
     }
    CloseHandle(m);
    }
   CloseHandle(f);
   printf(“%6d %s\n”, words,
    argv[i]);
   total += words;
   }
  }
 if (argc > 2)
 printf(“%6d Total\n”, total);
 fflush(stdout);
 ExitProcess(0);
 }
```

What you can see immediately is that there's a bit of a trade-off. Setting up the mapping of the file in the application's memory space involves a bit more code, but the inner-loop word counting is simplified by the absence of any further explicit I/O operations. Presumably, in most real-world situations (in contrast to this toy application), this trade-off will heavily favor the use of mapped files.

When I ran this second version, it was roughly as fast—maybe a couple of percentage points slower on small files. But on large files it was considerably faster. On nontrivial applications, that advantage for mapped files should increase.

The steps involved in mapping a file into memory consist of, first, opening it in the traditional manner; second, creating a mapping object; and, third, creating a view of that map. By creating a mapping object, you're telling the NT kernel's virtual memory subsystem that it should be prepared to allow the file to be mapped into an application's memory space. Documentation is a bit sketchy, but presumably this means setting up sufficient page-table entries to cover the file, initializing them so that if any process tries to access these pages the data will be faulted in and so on.

In this example, I've specified zeroes for the high and low 32-bit halves of the map object size, meaning the size is set at the current size of the file; but, for example, if I intended to increase the size of the file, I could have explicitly set a different size. It's also possible to name the mapping object so other processes might open it by name, though, in this example, I just specified NULL—making this a private mapping object.

Creating a view consists of making the mapped file visible in a particular application's memory space at a particular virtual address. (The virtual address at which it'll show up is chosen by the kernel; if you really cared, you could use the MapViewOfFileEx call instead.) Optionally, one might choose to map only a portion of a file by specifying the offset at which to begin and the number of bytes to map. In this example, by specifying zeros, I've made the whole file visible.

Multiple views can be created by multiple processes and still all be coherent, meaning they will always be consistent, one with another. The underlying mechanism here is that the same physical pages of memory representing a given portion of a file get shared between all the processes requesting access. The per-process tables mapping from virtual address to physical pages of memory will actually have the same entries in each case when two processes are viewing the same mapping object. If one process makes a change, it's seen instantly by the other process because it is physically the same memory cell seen by both.

(What you may already have guessed but which may bear explicit mention is that mapped files are also the way to do shared memory under Windows NT. Where other operating systems might have a special mechanism just for shared memory, NT gives you a more general mechanism that also happens to work nicely in that special case.)

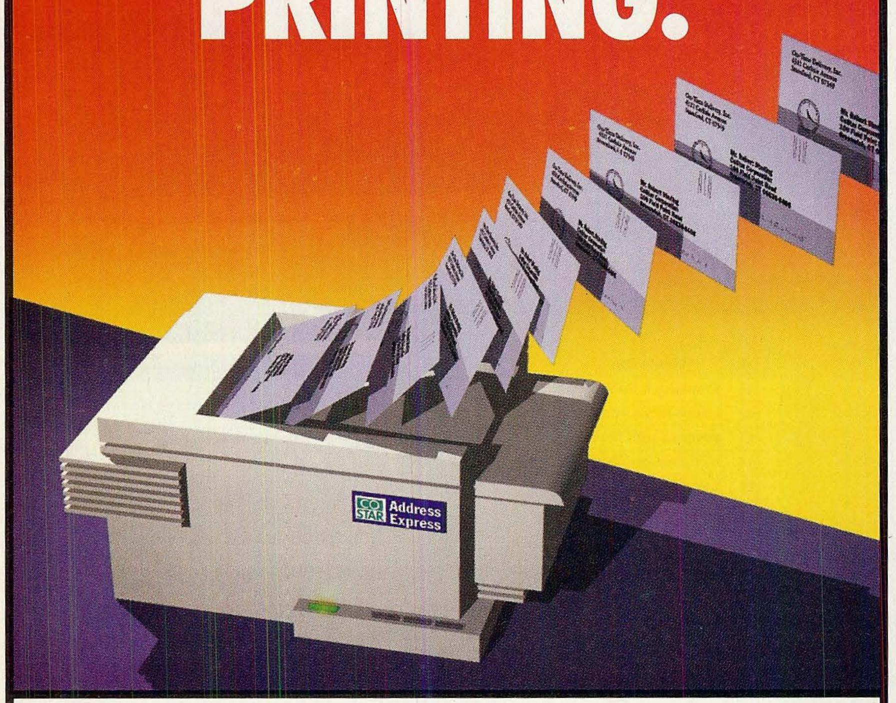
But do be aware that, even though individual views generated from a single map are guaranteed to be coherent, no such guarantee is made about views generated from different maps of the same file nor about the contents of a file as seen through a map versus ordinary I/O.

This example has dealt with just reading a file. It's also possible to write a file using a map. To do that, the file must be opened with GENERIC_READ | GENERIC_WRITE access, the map created with PAGE_READWRITE access and the view opened with FILE_MAP_WRITE access. If it's a new file, its size will be set by the length specified with the CreateFileMapping call, and any characters not set to some specific value will be zero. If you'd like to use a map to write a file whose size won't be known until all the writing is done, the trick is to start by creating a map bigger than you think is required, writing what you need, closing the map, closing the file and then reopening the file with GENERIC_WRITE access and setting the file size using SetFilePointer and SetEndOfFile. (For some reason, it's not possible to set the file pointer on a file opened for read/write access.)

Mapped files are not hard to use. And given the simplification and performance benefits they offer, they have a big advantage in complex applications. Mapped files are a sleeper: Because they're new and are offered only on a very few other systems, they haven't yet attracted a lot of attention. But stay tuned: Mapped files are going to be very important. ■

Douglas Hamilton is president of Hamilton Laboratories (Wayland, Mass.) and author of the Hamilton C Shell, an advanced interactive command processor and tools package for OS/2 and Windows NT. Reach Douglas on WIX as hamilton or care of Editor at the address on page 14.

EASY ENVELOPE PRINTING.



Introducing Address Express

Now there's a small inkjet printer just for envelopes, postcards, labels and self-mailers.

Address Express connects to any PC or Mac, is networkable, prints up to 4 envelopes per minute, and doesn't interfere with your main printer.

It comes with DOS, Windows and Mac software that *automatically* finds the address in your letter and prints an envelope while your main printer handles your letter. You can add return addresses, messages, graphics and logos, even POSTNET barcodes for faster mail delivery.

Call 1-800-426-7827, ext. 187 for more information.

CO STAR®

Tel: 203-661-9700 Fax: 203-661-1540 ©1993 CoStar Corporation.

Circle Reader Service No. 320