

## Re: Decoding strategy

---

*Source:*

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2006-10/msg02583.html>

---

- *From:* "Peter Duniho" <NpOeStPeAdM@xxxxxxxxxxxxxxxxxxxx>
  - *Date:* Fri, 13 Oct 2006 11:20:21 -0700
- 

<marcin.rzeznicki@xxxxxxxx> wrote in message  
<news:1160663537.474472.154060@xx>

That's the same, but under different names. CreateFileMapping reserves VM range.

That is incorrect. The virtual memory range is not reserved until you call MapViewOfFile.

[...] It is not yet committed, and you pay almost no resources—usage/performance price. MapViewOfFile commits some part of previously reserved VM and brings contents of file (maybe lazily, I don't know for sure)

That is also incorrect. MapViewOfFile reserves the virtual address space. There may be some caching, but otherwise committing the file data to physical RAM does not occur until a specific portion of the reserved virtual address space is referenced.

I'm offline right now, otherwise I'd provide a link to the MSDN web site. However, you can easily look those functions up yourself, and the documentation explicitly describes the behavior as I do above.

From the documentation for CreateFileMapping:

Creating a file mapping object creates the potential for mapping a view of the file, but does not map the view. The MapViewOfFile and MapViewOfFileEx functions map a view of a file into a process address space

If CreateFileMapping was what allocated virtual address space, it would not make sense for MapViewOfFileEx to even exist, since the main reason for that function is to allow the program to provide a specific virtual memory

## Re: Decoding strategy

address at which to map the file.

[...]

Well, actually, if I understand docs correctly, `CreateFileMap` reserves virtual memory address range and establishes association between VM addresses and file. `MapViewOfFile` brings contents of file to RAM

What can I say? You don't understand the docs correctly.

[...]

Positive with respect to "swapping" definition :-) It does not get swapped to swap file, true, but still it may be swapped to the mapped file. So, though you are right that memory pressure is removed from page file, you still pay the price of swapping if lot of RAM is occupied by file view

My point is that the amount of data in physical RAM will be related to your use of that data. The OS will keep the data in physical RAM based on your access of that data, not based on how much of it there is. This is true whether you use memory mapping or not.

With either technique, you can limit the \*maximum\* amount of physical RAM potentially consumed. Using memory mapping, you do this by mapping only a small range of the file at a time. Using conventional file i/o, you do this by limiting your own buffers that are used to store data you've read from the file.

In either case, the OS has the final say on how much physical RAM is actually used. Using memory mapping, if there are other depends on physical RAM, then only a portion of mapped virtual address space will actually be resident at any given time. Likewise, using conventional file i/o, only a portion of your own program buffers will be resident in physical RAM at any given time.

But memory mapped file i/o will not in and of itself increase memory swapping. The only way it could do that is if you not only map the entirety of a very large file in RAM, but you wind up \*accessing\* the totality of that file more frequently than you access anything else. In that case, the OS would be chasing you trying to keep all of the file data you're referencing resident, at the same time that other stuff needs to be swapped in and back out.

This is not a typical case, and doesn't seem relevant to your own situation. In any case, the OS is pretty smart. If your use of a memory mapped file starts pressuring other users of physical RAM, the OS is not going to bother trying to keep all of the memory mapped file in RAM. Even better, as long as you open the file as read-only, you're assured to never have to have the

## Re: Decoding strategy

cost of writing any data back to the disk if a physical page of RAM used by the file mapping has to get discarded and used for something else.

Your worries about memory mapping the entire file causing some serious problem with disk swapping are unfounded.

There is no reason that I can think of that would cause mapping a large file into virtual address space to cause any more swapping than processing that file would cause in any case. The OS certainly does not read all 500MB of a mapped 500MB file into physical RAM just because you've mapped the file.

I think that when I've established a view then RAM gets occupied. So, as I said, I map whole file at once as docs assure me that there is nothing wrong with that, but I restrict myself to moderately sized views.

But it's not true that when you establish a view then RAM gets occupied. The "view" is an allocation of virtual address space, not physical RAM.

That's not what I mean. If you were doing what I was suggesting already, then the only issue remaining for you would be figuring out when you need to back up in the data. The actual backing up would be trivial...you'd just decrement your pointer and read the byte you want to read. You would have moments when the mapped section of the file would have to change, but that would be a momentary diversion and you'd get right back to just reading the bytes from the mapped address space.

Sorry Peter, I don't get it then. Could you explain it to me, it seems to be interesting idea, but now I feel that I've got lost.

Assume you have some code that attempts to retrieve a byte from a specific file offset. Assume also that you have some code that translates this into access from your mapped view of the file. Finally, assume that the higher-level code is trying to access a byte that is just before the lowest file offset currently being mapped.

In pseudocode then:

Re: Decoding strategy

## Re: Decoding strategy

```
// The desired byte offset from the file
long ibFileOffset;
// This is the mapped range, "Min" inclusive, "Mac" exclusive
long ibMappedMin, ibMappedMac;
// The resulting offset within the mapped range
long ibMappedOffset;

if (ibFileOffset < ibMappedMin || ibFileOffset >= ibMappedMac)
{
// remap file so that ibMappedMin < ibFileOffset and
// ibFileOffset < ibMappedMac. Don't forget to make sure
// that ibMappedMin and ibMappedMac remain between 0 and
// the total file length.
}

ibMappedOffset = ibFileOffset - ibMappedMin;
return *(pbMappedData + ibMappedOffset);
```

Basically, in the normal case, all that the code is doing is translating the file offset to the mapping offset and returning the data at that offset. When the requested data falls outside the range, you just shift the offset enough to accomodate the new request for data.

Most likely, you'd try to center the newly-mapped range on the request file offset. When you get near the beginning or end of the file, you'll necessarily wind up at least trimming the mapped range as appropriate (making it smaller than normal), if not just pinning the range to the relevant boundary (preserving the total size of the mapping).

Isn't that what `ReadPage` in my code does? It is asked to bring contents indexed by block offset, it computes "real" offset and establishes a view. Decoder part does not even have to think of byte offsets because it operates on current page only, and pointer to it is constant in time when decoder operates.

IMHO, there's no reason for the decoder to have to think of pages within the file. As near as I can tell, that's an arbitrary choice affected by the implementation of your file i/o. In particular, if I understand correctly (and maybe I don't), part of the issue of "tearing" that you're worried about comes about because of the potential for data being read to cross one of these page boundaries.

The decoder should be concerning itself only with the entire file. That's why you have the tearing issue. If you allowed the decoder to simply use an offset relative to the beginning of the file, then the decoder would never have to worry about whether the data falls outside the currently mapped range. The i/o code would take care of that instead, and always return whatever byte it is the decoder wants to handle.

## Re: Decoding strategy

Of course, if you simply map the entire file all at once, the issue becomes trivial. So this may or may not be a moot point. You don't seem to be basing your architectural decisions on correct information about how file mapping works, so maybe understanding correctly how file mapping works you will find all of this "map a subset of the file" stuff becomes irrelevant.

[...]

So, if I understand correctly what you wrote, I am not concerned with mapping file at once, I reserve all VM I will need for one file (CreateFileMapping). But I am concerned when it comes to commit (MapViewOfFile) because that's where memory resources are really consumed. Am I missing something?

Yes, I think so. See above. :)

Pete

.