

Re: Decoding strategy

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2006-10/msg02222.html>

- *From:* marcin.rzeznicki@xxxxxxxx
 - *Date:* 11 Oct 2006 16:12:19 -0700
-

Peter Duniho napisal(a):

<marcin.rzeznicki@xxxxxxxx> wrote in message
news:1160597995.002328.207620@xx

[...]

So, memory mapped view is sure too be usable for a while, so I think it pays off too keep that in memory. That characteristic also ensures me that OS cache can be helpful and performance will not suffer from misses/disk reads very often.

That's well and good. However, those characteristics assist in ensuring that the file data is cached when using other forms of i/o as well, including using a FileStream. The benefit is not unique to memory mapped file i/o.

I see. I am on the winning side though, because I eliminate unnecessary in-memory copying. But, agreed, that may not be much overall.

Yes, I know. I was referring to something else. Sorry for being unclear. Docs say:

"(..)must specify an offset within the file that matches the memory allocation granularity of the system, or the function fails. That is, the offset must be a multiple of the allocation granularity".

I know that this is going to be aligned to sth in VM, but I do not care, this is transparent unless you write kernel-mode stuff or, generally, very low level stuff. What I do care is that I cannot choose FILE offset at wich mapping starts. And that leads to "tearing"

Okay, I think I understand better what you meant. I'm going to snip a bunch

Re: Decoding strategy

of stuff here, and hopefully jump to the core of the issue...

[...]

Well, that's very close to what I have now. Let me specify the details. I read few "blocks" a time, namely 4, which is 256kb of data (block for me is memory allocation granularity, as that it is the smallest addressable part of file when it comes to memory mapping). I try to adjust offset a little, so that: I always read the whole data I am requested, and, immediate reads in the neighbourhood will not cause remapping, which is close to your idea. But then, how do you know whether the very first byte of current "window" is the first block of character?

Thanks. The code you posted helps me understand better what's going on.

In fact, as near as I can tell, you are using memory mapping in practically the same way as my proposed multiple-buffer solution deals with things. That is, you're windowing the file with memory mapping the same way I'm doing it with the buffers.

Here's a dumb question: is there any particular reason you're NOT mapping the entire file at once? I've mentioned the possibility in previous messages, making assumptions that you have your reasons for not doing so. But if you could, all of these issues just go away. Are you genuinely concerned that you won't have enough contiguous virtual address space to map the whole file?

Well, there are two issues involved, and I do not know which one are you referring to. Let me explain. Mapping is actually two-step process, first of all you reserve VM for mapping and then you commit, which result in bringing contents of file to memory. So, when it comes to reservation step, I map entire file at once, code I pasted does not show this step. What is shown is the commitment step, and I commit only small portion of reserved memory at once. This app is not going to be server app, running on high end machines with many gigs of ram. It is rather intended to be desktop app. So, I do not want to reserve like 500 MB of memory for just one file because it could easily cause constant swapping and overall performance degradation on user machine.

Anyway, for the moment let's assume that you can only map a portion of the file at a time...

Depending on what the actual performance is, it seems to me that either method would be the correct solution. I suspect that the answer is to simply do the memory mapping a little differently, but I don't have enough experience with memory mapped files to know for sure.

Re: Decoding strategy

Specifically: what if you modified your code that maps the file, so that it maps a range *around* the starting point, the way I suggested with the buffers? At certain points (perhaps only when you got right to the very edge and attempted to read a byte outside your mapped range), you would remap the file, shifting the window so that the bytes you want to deal with are within the mapped range.

It does. Well, I am sorry, because I stripped this code of mapping logic, but when you see sth like `firstBufferIndex` it is, almost in all cases, carefully computed index of a portion which contains requested data but also its neighbourhood, so that near "jumps" should not cause remapping. Actually user may as well use enumerated acces, in that case I know in advance tha data is going to be read forward, then I can, if I must, map from where previous mapping ends.

When you index the data, I would recommend the high-level code using an index relative to the file beginning. That is, your index is just the file offset for the data (note that I'm not using the word "index" to relate to the broader index you calculate for the file data...I just mean a way to identify which byte you're working on at the moment). Then you translate that to the actual offset within the mapped range as necessary. That way, you can be changing the mapped range on the fly without affecting how the higher-level code that actually processes the data works.

Well, that is not going to work for me unfortunately. Interfaces I have to implement imply that data access uses "string coordinates" – so client code specifies – I want 5th char, not 5th byte, and reckoning that encoding–hell I would not be able to compute that easily, so I decided to use only "string coordinates".

I believe that performance should be fine doing this. When you remap the file, most of the file should still be in physical RAM and I suspect the OS will correctly reattach the newly mapped range to the portions of the range that are already resident in RAM. Only the newly mapped portions of the file should need to be read.

Yeah, I think so too

[...]

:--(That's pain in the ass for me. If I knew that I could always look back for missing parts of single character, then mix of your solution

Re: Decoding strategy

with memory mapping would be the best scheme

Well, at some point you need to come up with some mechanism for finding the beginning of a valid character. :) How you access the file might make this easier or harder, but the problem exists even if you can map the entire file at once. Sorry I can't be more helpful on that front. I agree, that part actually seems to be the "hard part" of this problem, in spite of all the space we've consumed discussing the file i/o part. :)

Yes, this problem vanishes when you are able to map AND decode entire file at once. But that's overkill I suppose.
So, I'll try to implement DecoderFallback, with nothing more than HOPE that it will always be able to do its job :-)
Thank you.

Pete