Why do I observe reads from a memory-mapped file when writing large blocks?

devblogs.microsoft.com/oldnewthing/20241107-00

November 7, 2024

A customer had created a memory-mapped file, and wanted to set large chunks of the memory to zero. For concreteness, consider this function that takes a memory-mapped file mapping (`map`) and a collection of blocks, each represented as a file offset and length. Its job is to set each block to zero.

```
void ZeroOutBlocks(uint8_t* map,
    std::initializer_list<
        std::pair<uint32_t, uint32_t>> blocks)
{
    for (auto&& block : blocks) {
        memset(map + block.first, 0, block.second);
    }
}
```

The customer found that even though this function is performing only write operations, a performance trace showed that the system was nevertheless reading from the file.

Memory-mapped files work by trapping the first access to each page. When the access occurs, the entire page is read from the disk, mapped into memory, and then the memory access operation is permitted to proceed.

The system does this regardless of whether the access was a read or a write. After all, that one written byte has to be merged with the existing content of the page.

“But in my case, the offsets and lengths are all page multiples, so there is no need to read the entire page into memory.”

Well, you know that, but the CPU doesn’t.

The CPU sees the first write to the page and traps to the kernel. The operating system doesn't go to the trouble of analyzing the code surrounding the fault and realizing that this code sequence:

```
@@: vmovntdq ymmword ptr [rcx],ymm0
    vmovntdq ymmword ptr [rcx+20h],ymm0
    vmovntdq ymmword ptr [rcx+40h],ymm0
    vmovntdq ymmword ptr [rcx+60h],ymm0
    vmovntdq ymmword ptr [rcx+80h],ymm0
    vmovntdq ymmword ptr [rcx+0A0h],ymm0
    vmovntdq ymmword ptr [rcx+0C0h],ymm0
    vmovntdq ymmword ptr [rcx+0E0h],ymm0
    add     rcx,100h
    sub     r8,100h
    cmp     r8,100h
    jae     @B
```

is a `memset` loop that writes `r8` ÷ 256 × 8 copies of the `ymm0` register to consecutive bytes of memory.

But suppose the operating system had code to recognize the top 50 most common implementations of `memset`. And suppose that it saw that the `memset` was going to write an entire page of zeroes. In that case, could it avoid the useless read?

I guess the operating system could do that. It would have to realize that this was a full-page memset and perform the same memset on the newly-mapped page before making the memory visible to the process (in case other threads read from the page before the faulting thread finishes the loop).

Still, that's a lot of `memset` detection, since it would have to run at every write page fault. I suspect the write page faults that are due to `memset` of more than one page represent too small a fraction of total write page faults to be worth the trouble.

But who knows. My intuition on this has been wrong before.

Update: Commenter Nir Lichtman pointed out that memory-mapped files and I/O are not necessarily coherent, so the `WriteFile` trick doesn't work.

~~What you can do instead is write zeroes to the memory-mapped file the old-fashioned way: with `WriteFile`. Since Windows NT unifies memory-mapped files with the file cache, these writes will be coherent with the memory mapping. If you want to get fancy, you can use overlapped writes and wait for them all to complete.~~

~~You need only set aside one page of zeroes for this trick. For regions up to a page in size, you can issue a `WriteFile` from your special zero-buffer. Regions larger than a page can be broken up into page-sized chunks, or you can use `WriteFileGather` to write that page to consecutive pages in the file.~~