What happens to the contents of a memory-mapped file when a process is terminated abnormally?

devblogs.microsoft.com/oldnewthing/20100428-00

April 28, 2010

Raymond Chen

Bart wonders what happens to the dirty contents of a memory-mapped file when an application is terminated abnormally.

From the kernel's point of view, there isn't much difference between a normal and an abnormal termination. In fact, the last thing that `ExitProcess` does is `Terminate-Process(GetCurrentProcess(), ExitCode)`, so in a very real sense the two operations are identical from the kernel's point of view. The only difference is that in a controlled termination, DLLs get their `DLL_PROCESS_DETACH` notifications, whereas in an abnormal termination, they don't. But given that the advice for DLLs is to do as little as possible during process termination (including suppressing normal cleanup), the difference even there is negligible.

Therefore, the real question is *What happens to the dirty contents of a memory-mapped file when an application exits without closing the handle?*

If a process exits without closing all its handles, the kernel will close them on the process's behalf. Now, in theory, the kernel could change its behavior depending on why a handle is closed—skipping some steps if the handle is being closed as part of cleanup and performing additional ones if it came from an explicit `CloseHandle` call. So it's theoretically possible that the unwritten memory-mapped data may be treated differently. (Although it does violate the principle of not keeping track of information you don't need. But as we've seen, sometimes you have to violate a principle.)

But there's also the guarantee that multiple memory-mapped views of the same local file are *coherent*; that is, that changes made to one view are immediately reflected in other views. Therefore, if there were another view of that memory-mapped file which you neglected to close manually, any changes you had made to that view would still be visible in other views, so the contents were not lost. It's not like the kernel is going to fire up its time machine and say, "Okay, those writes to the memory-mapped file which this terminated application made, I'm going to go back and undo them even though I had already shown them to other applications."

In other words, in the case where the memory-mapped view is to a local file, and there happens to be another view on the file, then the changes are not discarded, since they are being kept alive by that other view.

Therefore, if the kernel were to discard unflushed changes to the memory-mapped view, it would have to have not one but two special-cases. One for the “this handle is being closed implicitly due to an application exiting without closing all its handles” case and another for the “this handle being closed implicitly due to an application exiting without closing all its handles when the total number of active views is less than two.”

I don’t know what the final answer is, but if the behavior were any different from the process closing the handle explicitly, it would require two special-case behaviors in the kernel. I personally consider this unlikely. Certainly if I were writing an operating system, I wouldn’t bother writing these two special cases.

If you think like the memory manager, then you come to the same conclusion from a different direction. If you think about the lifetime of a committed page, there are a small set of operations each page type needs to perform.

- *Page in*: Produce the contents of the page.
- *Make dirty*: The page has been written to for the first time.
- *Page out dirty*: The page is about to be removed from memory. The application has written to the page since it was paged in.
- *Page out clean*: The page is about to be removed from memory. The application has not written to the page since it was paged in.
- *Decommit dirty*: The page is about to be freed and it was written to since it was paged in.
- *Decommit clean*: The page is about to be freed and it was not written to since it was paged in.

The different types of committed pages implement these operations in different ways. Because, after all, that’s what makes them different.

- Zero-initialized memory
    - *Page in*: Fill the page with zeroes.
    - *Make dirty*: Locate a free page in the swap file, assign it to this page, set type to “allocated memory”.
    - *Page out dirty*: (never happens)
    - *Page out clean*: Do nothing.
    - *Decommit dirty*: (never happens)
    - *Decommit clean*: Do nothing.

- Allocated memory
    - *Page in*: Read page contents from swap file.
    - *Make dirty*: Do nothing.
    - *Page out dirty*: Write page contents to swap file.
    - *Page out clean*: Do nothing.
    - *Decommit dirty*: Free the page from the swap file.
    - *Decommit clean*: Free the page from the swap file.
- Memory-mapped file
    - *Page in*: Read page contents from file.
    - *Make dirty*: Do nothing.
    - *Page out dirty*: Write page contents to file.
    - *Page out clean*: Do nothing.
    - *Decommit dirty*: Write page contents to file.
    - *Decommit clean*: Do nothing.

There are other types of pages (such as *copy-on-write pages*, the *null page*, and *physical pages*, but they aren't relevant here.)

Note that these operations apply to the pages and not to the address space. Memory can be committed without being visible in the address space, and a single page can be visible in multiple address spaces at once, or even multiple times within the same address space! The reason two views onto the same local file are coherent is that they are merely two manifestations of the same underlying committed page. The part of the memory manager that manages committed memory doesn't know where in the address space (if anywhere) the memory is going to be mapped, nor does it know why the requested operation is taking place (beyond the circumstances implied by the operation itself).

When a memory-mapped file page is decommitted, the appropriate *Decommit function* is called, and if the page is dirty, then the contents are flushed to the underlying file. It doesn't know why the decommit happened, so it can't perform any special shortcuts depending on the circumstances that led to the decommit.

Consider a memory-mapped file with two views. One view closes normally. The page is still committed (the second view is still using it), so no *Decommit* happens yet. Then the process which was using the second view terminates abnormally. The *Decommit* must still be treated as a normal (not abnormal) decommit, because the first process did terminate normally, and therefore is under the not unreasonable expectation that its changes will make it into the file. In order to protect against discarding changes which earlier (now-closed) views had made, an extra bit would have to be carried for each committed page that says, "This page contains data that we promised to write back to the file (because somebody wrote to it and then closed the view normally)." You would set this flag on every page in a view when you close the view normally, or if you close the view due to abnormal process termination if there are other still-running processes that are using the view (because the changes are visible to them), and you would clear this flag after each *Page out* operation. Then you could add another type of decommit, *Decommit leaked*, which is used when a page that contains no changes from properly-closed views is decommitted because the last remaining reference to it was from a process that terminated abnormally.

You could do all this work in your memory manager, but why bother? It’s additional bookkeeping just to optimize the case where somebody is doing something wrong.

Raymond Chen

Follow