How do FILE_FLAG_SEQUENTIAL_SCAN and FILE_FLAG_RANDOM_ACCESS affect how the operating system treats my file?

devblogs.microsoft.com/oldnewthing/20120120-00

January 20, 2012

Raymond Chen

There are two flags you can pass to the `CreateFile` function to provide hints regarding your program's file access pattern. What happens if you pass either of them, or neither?

Note that the following description is not contractual. It's just an explanation of the current heuristics (where "current" means "Windows 7"). These heuristics *have changed* at each version of Windows, so consider this information as a tip to help you choose an appropriate access pattern flag in your program, not a guarantee that the cache manager will behave in a specific way if you do a specific thing.

If you pass the `FILE_FLAG_SEQUENTIAL_SCAN` flag, then the cache manager alters its behavior in two ways: First, the amount of prefetch is doubled compared to what it would have been if you hadn't passed the flag. Second, the cache manager marks as available for re-use those cache pages which lie entirely behind the current file pointer (assuming there are no other applications using the file). After all, by saying that you are accessing the file sequentially, you're promising that the file pointer will always move forward.

At the opposite extreme is `FILE_FLAG_RANDOM_ACCESS`. In the random access case, the cache manager performs no prefetching, and it does not aggressively evict pages that lie behind the file pointer. Those pages (as well as the pages that lie ahead of the file pointer which you already read from or wrote to) will age out of the cache according to the usual most-recently-used policy, which means that heavy random reads against a file will not pollute the cache (the new pages will replace the old ones).

In between is the case where you pass neither flag.

If you pass neither flag, then the cache manager tries to detect your program's file access pattern. This is where things get weird.

If you issue a read that begins where the previous read left off, then the cache manager performs some prefetching, but not as much as if you had passed `FILE_FLAG_SEQUENTIAL_SCAN`. If sequential access is detected, then pages behind the file pointer are also evicted from the cache. If you issue around six reads in a row, each of which begins where the previous one left off, then the cache manager switches to `FILE_FLAG_SEQUENTIAL_SCAN` behavior for your file, but once you issue a read that no longer begins where the previous read left off, the cache manager revokes your temporary `FILE_FLAG_SEQUENTIAL_SCAN` status.

If your reads are not sequential, but they still follow a pattern where the file offset changes by the same amount between each operation (for example, you seek to position 100,000 and read some data, then seek to position 150,000 and read some data, then seek to position 200,000 and read some data), then the cache manager will use that pattern to predict the next read. In the above example, the cache manager will predict that your next read will begin at position 250,000. (This prediction works for decreasing offsets, too!) As with auto-detected sequential scans, the prediction stops as soon as you break the pattern.

Since people like charts, here's a summary of the above in tabular form:

Access pattern	Prefetch	Evict-behind
Explicit random	No	No
Explicit sequential	Yes (2×)	Yes
Autodetected sequential	Yes	Yes
Autodetected very sequential	Yes (2×)	Yes
Autodetected linear	Yes	?
None	No	?

There are some question marks in the above table where I'm not sure exactly what the answer is.

Note: These cache hints apply only if you use `ReadFile` (or moral equivalents). Memory-mapped file access does not go through the cache manager, and consequently these cache hints have no effect.

Raymond Chen

Follow