File I/O

Normally, file I/O goes through the filesystem cache, which has some performance benefits: caching of recently accessed data, read-ahead (speculative pre-fetching data from disk), write-behind (asynchronously writing data to disk), and combining of small writes. By hinting Windows about your expected file access pattern, you can gain a bit more performance. If your application does overlapped I/O and can intelligently handle buffers facing some complexities, then bypassing caching entirely can be more efficient.

Cache Hinting

When creating or opening a file, you specify flags and attributes to the `CreateFile` Win32 API function, some of which influence caching behavior:

- `FILE_FLAG_SEQUENTIAL_SCAN` hints to the Cache Manager that the file is accessed sequentially, possibly skipping some parts, but is seldom accessed randomly. The cache will read further ahead.
- `FILE_FLAG_RANDOM_ACCESS` hints that the file is accessed in random order, so the Cache Manager reads less ahead, since it is unlikely that this data will actually be requested by the application.
- `FILE_ATTRIBUTE_TEMPORARY` hints that the file is temporary, so flushing writes to disk (to prevent data loss) can be delayed.

NET exposes these options (except the last one) through a `FileStream` constructor overload that accepts a `FileOptions` enumeration parameter.

Caution Random access is bad for performance, especially on disk media since the read/write head has to physically move. Historically, disk throughput has improved with increasing aereal storage density, but latency has not. Modern disks can intelligently (taking disk rotation into account) re-order random-access I/O so that the total time the head spends travelling is minimized. This is called Native Command Queuing (NCQ). For this to work effectively, the disk controller must be aware of several I/O requests ahead. In other words, if possible, you should have several asynchronous I/O requests pending.

Unbuffered I/O

Unbuffered I/O bypasses the Windows cache entirely. This has both benefits and drawbacks. Like with cache hinting, unbuffered I/O is enabled through the “flags and attributes” parameter during file creation, but .NET does not expose this functionality:

- `FILE_FLAG_NO_BUFFERING` prevents the data read or written from being cached but has no effect on hardware caching by the disk controller. This avoids a memory copy (from user buffer to cache) and prevents cache pollution (filling the cache with useless data at the expense of more important data). However, reads and writes must obey alignment requirements. The following parameters must be aligned to or have a size that is an integer multiple of the disk sector size: I/O transfer size, file offset, and memory buffer address. Typically, the sector size is 512 bytes long. Recent high-capacity disk drives have 4,096 byte sectors (known as “Advanced Format”), but they can run in a compatibility mode that emulates 512 byte sectors (at the expense of performance).
- `FILE_FLAG_WRITE_THROUGH` instructs the Cache Manager to flush cached writes (if `FILE_FLAG_NO_BUFFERING` is not specified) and instructs the disk controller to commit writes to physical media immediately, rather than storing them in the hardware cache.

Read-ahead improves performance by keeping the disk utilized, even if the application makes synchronous reads, and has delays between reads. This depends on Windows correctly predicting what part of the file the application will request next. By disabling buffering, you also disable read-ahead and should keep the disk busy by having multiple overlapped I/O operations pending.

Write-behind also improves performance for applications making synchronous writes by giving the illusion that disk-writes finish very quickly. The application can better utilize the CPU, because it blocks for less time. When you disable buffering, writes complete in the actual amount of time it takes to write them to disk. Therefore, doing asynchronous I/O becomes even more important when using unbuffered I/O.