How to obtain good concurrent read performance from disk

Asked 16 years, 4 months ago Modified 9 years, 8 months ago

Viewed 10k times

27

I'd like to ask a question then follow it up with my own answer, but also see what answers other people have.

We have two large files which we'd like to read from two separate threads concurrently. One thread will sequentially read fileA while the other thread will sequentially read fileB. There is no locking or communication between the threads, both are sequentially reading as fast as they can, and both are immediately discarding the data they read.

Our experience with this setup on Windows is very poor. The combined throughput of the two threads is in the order of 2-3 MiB/sec. The drive seems to be spending most of its time seeking backwards and forwards between the two files, presumably reading very little after each seek.

If we disable one of the threads and temporarily look at the performance of a single thread then we get much better bandwidth (~45 MiB/sec for this machine). So clearly the bad two-thread performance is an artefact of the OS disk scheduler.

Is there anything we can do to improve the concurrent thread read performance? Perhaps by using different APIs or by tweaking the OS disk scheduler parameters in some way.

Some details:

The files are in the order of 2 GiB each on a machine with 2GiB of RAM. For the purpose of this question we consider them not to be cached and perfectly defragmented. We have used defrag tools and rebooted to ensure this is the case.

We are using no special APIs to read these files. The behaviour is repeatable across various bog-standard APIs such as Win32's CreateFile, C's fopen, C++'s std::ifstream, Java's FileInputStream, etc.

Each thread spins in a loop making calls to the read function. We have varied the number of bytes requested from the API each iteration from values between 1KiB up to 128MiB. Varying this has had no effect, so clearly the amount the OS is physically reading after each disk seek is not dictated by this number. This is exactly what should be expected.

The dramatic difference between one-thread and two-thread performance is repeatable across Windows 2000, Windows XP (32-bit and 64-bit), Windows Server 2003, and also with and without hardware RAID5.

windows multithreading file-io

Share

Improve this question

Follow

edited Oct 10, 2008 at 11:06

tzot

95.8k ● 30 ● 149 ● 208

asked Aug 12, 2008 at 19:50

pauldoo

18.6k ● 20 ● 97 ● 118

6 Answers

Sorted by: Highest score (default)

13

The problem seems to be in Windows I/O scheduling policy. According to what I found here there are many ways for an O.S. to schedule disk requests. While Linux and others can choose between different policies, before Vista Windows was locked in a single policy: a FIFO queue, where all requests where splitted in 64 KB blocks. I believe that this policy is the cause for the problem you are experiencing: the scheduler will mix requests from the two threads, causing continuous seek between different areas of the disk.
Now, the good news is that according to here and here, Vista introduced a smarter disk scheduler, where you can set the priority of your requests and also allocate a minimum badwidth for your process.
The bad news is that I found no way to change disk policy or buffers size in previous versions of Windows. Also, even if raising disk I/O priority of your process will boost

the performance against the other processes, you still have the problems of your threads competing against each other.
What I can suggest is to modify your software by introducing a self-made disk access policy.
For example, you could use a policy like this in your thread B (similar for Thread A):

```
if THREAD A is reading from disk then wait for
THREAD A to stop reading or wait for X ms
Read for X ms (or Y MB)
Stop reading and check status of thread A again
```

You could use semaphores for status checking or you could use perfmon counters to get the status of the actual disk queue. The values of X and/or Y could also be auto-tuned by checking the actual trasfer rates and slowly modify them, thus maximizing the throughtput when the application runs on different machines and/or O.S. You could find that cache, memory or RAID levels affect them in a way or the other, but with auto-tuning you will always get the best performance in every scenario.

Share Improve this answer Follow

answered Aug 13, 2008 at 13:47

Andrea Bertani
1,637 ● 12 ● 11

6

I'd like to add some further notes in my response. All other non-Microsoft operating systems we have tested do not suffer from this problem. Linux, FreeBSD, and Mac OS X (this final one on different hardware) all degrade much more gracefully in terms of aggregate bandwidth when moving from one thread to two. Linux for example degraded from ~45 MiB/sec to ~42 MiB/sec. These other operating systems must be reading larger chunks of the file between each seek, and therefor not spending nearly all their time waiting on the disk to seek.

Our solution for Windows is to pass the `FILE_FLAG_NO_BUFFERING` flag to `CreateFile` and use large (~16MiB) reads in each call to `ReadFile`. This is suboptimal for several reasons:

- Files don't get cached when read like this, so there are none of the advantages that caching normally gives.
- The constraints when working with this flag are much more complicated than normal reading (alignment of read buffers to page boundaries, etc).

(As a final remark. Does this explain why swapping under Windows is so hellish? Ie, Windows is incapable of doing IO to multiple files concurrently with any efficiency, so while swapping all other IO operations are forced to be disproportionately slow.)

Edit to add some further details for Will Dean:

Of course across these different hardware configurations the raw figures did change (sometimes substantially). The problem however is the consistent degradation in performance that only Windows suffers when moving from one thread to two. Here is a summary of the machines tested:

- Several Dell workstations (Intel Xeon) of various ages running Windows 2000, Windows XP (32-bit), and Windows XP (64-bit) with single drive.
- A Dell 1U server (Intel Xeon) running Windows Server 2003 (64-bit) with RAID 1+0.
- An HP workstation (AMD Opteron) with Windows XP (64-bit), and Windows Server 2003, and hardware RAID 5.
- My home unbranded PC (AMD Athlon64) running Windows XP (32-bit), FreeBSD (64-bit), and Linux (64-bit) with single drive.
- My home MacBook (Intel Core1) running Mac OS X, single SATA drive.
- My home Koolu PC running Linux. Vastly underpowered compared to the other systems but I demonstrated that even this machine can outperform a Windows server with RAID5 when doing multi-threaded disk reads.

CPU usage on all of these systems was very low during the tests and anti-virus was disabled.

I forgot to mention before but we also tried the normal Win32 `CreateFile` API with the `FILE_FLAG_SEQUENTIAL_SCAN` flag set. This flag didn't fix the problem.

Share Improve this answer Follow

edited Apr 20, 2015 at 8:26

wilx
18.2k ● 7 ● 64 ● 120

answered Aug 12, 2008 at 19:51

pauldoo
18.6k ● 20 ● 97 ● 118

Welcome to Microsoft. – v.oddou Aug 4, 2015 at 2:42

1

It does seem a little strange that you see no difference across quite a wide range of windows versions and nothing between a single drive and hardware raid-5.

It's only 'gut feel', but that does make me doubtful that this is really a simple seeking problem. Other than the OS X and the Raid5, was all this tried on the same machine - have you tried another machine? Is your CPU usage basically zero during this test?

What's the shortest app you can write which demonstrates this problem? - I would be interested to try it here.

Share Improve this answer Follow

answered Aug 12, 2008 at 20:20

Will Dean
39.5k ● 11 ● 92 ● 118

per the single-drive vs raid5: if reading sequential data from two large enough files, you can't avoid *all* the disk heads seeking back and forth; the stripe size is typically 16-128kiB, so to read 1MiB of data, you need all (or most) of the heads to seek there. – tzot Oct 10, 2008 at 11:23

0

I would create some kind of in memory thread safe lock. Each thread could wait on the lock until it was free. When the lock becomes free, take the lock and read the file for a defined length of time or a defined amount of data, then release the lock for any other waiting threads.

Share Improve this answer Follow

answered Aug 12, 2008 at 20:04

Stacey Richards
6,596 ● 7 ● 39 ● 40

0

Do you use IOCompletionPorts under Windows? Windows via C++ has an in-depth chapter on this subject and as luck would have it, it is also available on MSDN.

Share Improve this answer Follow

answered Aug 12, 2008 at 21:38

graham.reeds
16.5k ● 17 ● 75 ● 138

0

Paul - saw the update. Very interesting.

It would be interesting to try it on Vista or Win2008, as people seem to be reporting some considerable I/O improvements on these in some circumstances.

My only suggestion about a different API would be to try memory mapping the files - have you tried that? Unfortunately at 2GB per file, you're not going to be able to map multiple whole files on a 32-bit machine, which means this isn't quite as trivial as it might be.

Share Improve this answer Follow

answered Aug 13, 2008 at 9:44

Will Dean
39.5k ● 11 ● 92 ● 118

going to such extends just to make something work on windows, I would simply advocate to move his process to linux. what is the cost of each solution ? honestly... – v.oddou Aug 4, 2015 at 2:43