Fast writing: Memory-Mapped file versus BufferedWriter

Has anyone benchmarked this? I want to write as fast as possible to disk, minimizing the latency of my write calls. I wonder if writing to a memory mapped buffer (through buffer.put()) is faster than just buffering the contents in the Java side and just flushing to the fileChannel once the buffer is full. That way I would be just making a system call (FileChannel.write) once the buffer gets full. I am not sure what happens when I write some bytes to the MappedByteBuffer, in other words, if a system call is done or not.

With the buffered approach, I would be able to write to the disk in chunks of 16,32 or 64k, which I believe is optimum. The drawback of buffering is that you must have a shutdown hook to flush the contents if the application crashes. And an extra layer of copying to the buffer instead of writing straight to the file channel. And if you tail the file you may not see what you want because the contents are buffered. I also have no clue what happens if you tail a memory mapped file.

Any experienced soul can help out here?

java file-io io real-time nio

edited Oct 21 '12 at 15:36 asked Oct 21 '12 at 15:27
TraderJoeChicago **2,857** 4 33 46

I haven't benchmarked this, so this is a comment, but from everything I've read memory mapped IO is *supposed* to be much faster. – Chris Thompson Oct 21 '12 at 15:47

1 Answer

If you are trying to minimise latency, I have found writing to MemoryMappedFiles to be faster because it *avoids* the need to make a system call. This assumes you don't need to force the data to disk and are happy for the OS to do this on a best effort basis.

The typical latency for writing to a MemoryMappedFile is the same as writing to memory, so i don't believe you will get faster. As the file grows you need to perform an additional memory mapped regions and this can take 50 to 100 micro-seconds which is significant but should be rare enough that it doesn't matter.

Writing to IO via a system call takes in the order of 5 to 10 micro-seconds which is fast enough for more applications, but relatively much slower if it matters.

If you have a need to see the data as it is written with a low latency, I suggest you look at my library Java Chronicle which supports reading data with a typical latency of 100 ns from the time it is written.

Note: While memory mapped files can reduce latency of individual writes it doesn't increase the write throughput of your disk subsystem. This means that if you have a slow disk sub-system, your memory will soon become exhausted (even if you have many GBs) and this will be the performance bottleneck regardless of which approach you take.

For example if you have SATA or fibre, you might have a limit of 500 MB/s which is easy to exceed in which case once you hit you memory limit, this will slow you down regardless of what you chose.

edited Oct 21 '12 at 17:33 answered Oct 21 '12 at 17:20
Peter Lawrey **314k** 31 345 639

Thanks, Peter. My need is low latency but I don't care too much about throughput. It looks in theory that writing to a memory mapped file would be as fast as writing to another thread for asynch logging, or even faster because there is no need for synchronization, but in practice I have seen that async logging is way faster. Maybe because my non-blocking queue is ultra-fast but I feel I need to do more benchmarks on that matter. One thing I am doing is pausing for a number of ns before each log to lower the throughput. That made it much faster (~300ns). – TraderJoeChicago Oct 21 '12 at 21:56

I would be interested to see your results. I see less than 200 ns round trip from process A to process B back to process A and if you have found something faster than that I would like to know. – Peter Lawrey Oct 22 '12 at 7:21

@TraderJoeChicago: Memory mapping must occasionally close the mapping, extend the file, and reopen the mapping, all synchronously. A background thread avoids this. I'd guess that's the difference. – Mooing Duck Jan 22 '14 at 16:55