Fast File IO with .NET 6

Adam Sitnik

What are you going to learn from this talk?

- What are all FileStream capabilities and how to use them.
- How to open files for 100% async IO.
- What are the differences between Windows and Unix.
- What was changed in .NET 6.
- What APIs were introduced in .NET 6.
- How to get the best performance using available APIs.

Introduction to FileStream

- public **FileStream**(string path, FileMode mode, FileAccess access, FileShare share, int bufferSize, FileOptions options)
- public **FileStream**(SafeFileHandle handle, FileAccess access, int bufferSize, bool isAsync)
- The simple part:
  - mode: open, create, replace, truncate or open for appending
  - access: reading, writing, or both

string path

- relative or absolute path to a file. The file can be:
  - A regular file (the most common use case)
  - Symbolic link – it gets dereferenced
  - Pipe or a socket
    - Windows (#54676): \\.\pipe\namedPipeName
    - Unix: was already supported
  - Character or a block file
    - Unix: "/dev/tty", "/dev/console"
  - Device:
    - Windows (#54673): \\?\Volume{724edb31-eaa5-4728-a4e3-f2474fd34ae2}\
  - A network file:
    - //$machineName/$folderName/$filename
    - \\$machineName\$folderName\$fileName

FileShare

- File locks are mandatory on Windows and advisory on Unix.
- There is no 1:1 mapping:

FileShare	Windows	Unix
None (default)	Exclusive access	Exclusive access
Read	Others can read	Others can’t get an exclusive access
Write	Others can write	Others can’t get an exclusive access
Delete	Others can delete the file	Others can’t get an exclusive access

- .NET 6:
  - don’t acquire shared locks when writing to NFS/CIFS/SMB (#55256)
  - File locks can be disabled on Unix:
    - App context switch: System.IO.DisableFileLocking
    - Env var: DOTNET_SYSTEM_IO_DISABLEFILELOCKING

int bufferSize

- bufferSize sets the size of the FileStream **private buffer** used for **buffering.**
- Example:
  - User requests a read of **n** bytes.
  - If **n < bufferSize**, FS fetches bufferSize-many bytes from the OS. Stores them in it's private buffer and returns only **n** bytes.
  - The next read operation is going to return from the remaining buffered bytes and ask the OS for more, only if needed.
- This performance optimization allows FileStream to reduce the number of expensive sys-calls, as copying bytes is simply cheaper.
- The default value is 4096 (it's enabled).
- It can be disabled by setting it to 1 (every .NET) or 0 (#52928, .NET 6+).

Perf tip #1: Do you need buffering?

- What are your file access patterns? Do you read some metadata first or just copy data to memory buffers?
- What is your target environment? OS? File System? Typical file size?
- Run some benchmarks and **measure** if you need buffering or not.
- You can increase the bufferSize at a cost of increased memory allocations.
- If you disable it, you avoid some overhead. The overhead is non-trivial for async methods.

Buffering overhead for async file IO

```
private const string FilePath = "file.data";
private byte[] _userBuffer;

[GlobalSetup]
public void Setup()
{
    _userBuffer = new byte[16_000];
    File.WriteAllBytes(FilePath, new byte[100_000_000]); // 100 MB
}

[Benchmark(Baseline = true)]
public async Task Enabled()
{
    using FileStream fs = new FileStream(FilePath, FileMode.Open,
       FileAccess.Read, FileShare.None, bufferSize: 4096, true);

    while (await fs.ReadAsync(_userBuffer) != 0) ;
}

[Benchmark]
public async Task Disabled()
{
    FileStream fs = new FileStream(FilePath, FileMode.Open,
       FileAccess.Read, FileShare.None, bufferSize: 1, true);

    while (await fs.ReadAsync(_userBuffer) != 0) ;
}
```

Method	Mean	Ratio	Allocated
Enabled	100.64 ms	1.00	2 KB
Disabled	87.04 ms	**0.87**	1 KB

bool isAsync

- Allows for controlling whether the file should be opened for asynchronous or synchronous IO.
- The default value is **false**, which translates to **synchronous** IO.
- What if you specify isAsync=false and later use the *Async methods?
  - They are going to perform sync IO on a ThreadPool Thread.
  - No cancellation support.
- As of today (.NET 6) isAsync does not matter on Unix, where there is no good async file IO mechanism.
- It may change for Linux in the near future: io_uring (#51985).

Perf tip #2: do you need async IO?

- Do you need scalability? Cancellation support?
- Are you writing async code? It should be 100% async from the top to the bottom.
- If not, don’t specify isAsync=true. Just keep the defaults.
- If yes, then specify isAsync=true and don’t use the sync methods as they have non-trivial overhead for async file handles.

Sync IO using sync and async file handles

```
private const string FilePath = "file.data";
private byte[] _userBuffer;

[GlobalSetup]
public void Setup()
{
    _userBuffer = new byte[16_000];
    File.WriteAllBytes(FilePath, new byte[100_000_000]); // 100 MB
}

[Benchmark(Baseline = true)]
public void Read()
{
    using FileStream fs = new FileStream(FilePath, FileMode.Open, FileAccess.Read, FileShare.None, 1, isAsync: false);
    while (fs.Read(_userBuffer) != 0) ;
}

[Benchmark]
public void Read_AsyncHandle()
{
    using FileStream fs = new FileStream(FilePath, FileMode.Open, FileAccess.Read, FileShare.None, 1, isAsync: true);
    while (fs.Read(_userBuffer) != 0) ;
}
```

Method	Mean	Ratio
Read	27.24 ms	1.00
Read_AsyncHandle	114.89 ms	**4.22**

We have reduced the overhead (#54266)

Method	Runtime	Mean	StdDev	Ratio	Allocated
Read_AsyncHandle	.NET 5.0	124.28 ms	1.754 ms	1.00	**2 MB**
Read_AsyncHandle	.NET 6.0	61.33 ms	0.878 ms	**0.49**	**1 MB**

But it is still a bad idea!

[Flags] enum FileOptions

- Asynchronous – enables async IO (isAsync=true).
- SequentialScan, RandomAccess – hints for the OS.
- DeleteOnClose – usefull for simplifying file cleanup.
- WriteThrough – write directly to the device.

Perf tip #3: do you need to specify hints to OS?

- Modern kernels are good at recognizing file access patterns.
- On Unix, we need an additional sys-call to apply the hints.
- You need to measure if they provide actual benefit for your code.
- Most probably not ;)

FileStream(SafeFileHandle)

- User opens the file on their own and provides handle instead of path.
- In .NET 6 we have added a new API for opening file handles:

```
namespace System.IO
{
    public static class File
    {
        public static SafeFileHandle OpenHandle(string path,
                            FileMode mode = FileMode.Open,
                            FileAccess access = FileAccess.Read,
                      FileShare share = FileShare.Read,
                            FileOptions options = FileOptions.None,
                            long preallocationSize = 0)
```

Perf tip #4: FileStream.SafeFileHandle is expensive

- It performs an **additional sys-call**.
- **It always has side-effects**:
  - Prior to .NET 6: expensive offset checks are enabled.
  - .NET 6: syncing file offset with the OS.
- Prefer new File.OpenHandle over FileStream.SafeFileHandle.
- If you have to use FileStream.SafeFileHandle, do it once and store the handle. Avoid accessing it in a loop!

Summary: basics

- You can open any kind of files using FileStream and path.
- File locking works differently on Unix.
- Buffering might have non-trivial overhead. Do you need it?
- To get 100% async code specify FileStream(isAsync=true).
- Don’t use sync methods of FileStream opened for async IO.
- Try to avoid using FileStream.SafeFileHandle as it’s expensive. Use the new File.OpenHandle API.

Position tracking

- So far FileStream opened for async IO was synchronizing file offset with Windows for **every** async read or write.
- A blog post from the Windows Server Performance Team calls the API that allows for doing that an *anachronism*:

The old DOS SetFilePointer API is an anachronism. One should specify the file offset in the overlapped structure even for synchronous I/O. It should never be necessary to resort to the hack of having private file handles for each thread.

- Now the offset is tracked only in memory.
- We use sys-calls that require us to provide it in an explicit way.

Seek & Position

```
[Benchmark]
[Arguments(OneKibibyte, FileOptions.None)]
[Arguments(OneKibibyte, FileOptions.Asynchronous)]
public void SeekForward(long fileSize, FileOptions options)
{
    string filePath = _sourceFilePaths[fileSize];
    using (FileStream fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read, FileShare.Read, FourKibibytes, options))
    {
        for (long offset = 0; offset < fileSize; offset++)
        {
            fileStream.Seek(offset, SeekOrigin.Begin);
        }
    }
}

[Benchmark]
[Arguments(OneKibibyte, FileOptions.None)]
[Arguments(OneKibibyte, FileOptions.Asynchronous)]
public void SeekBackward(long fileSize, FileOptions options)
{
    string filePath = _sourceFilePaths[fileSize];
    using (FileStream fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read, FileShare.Read, FourKibibytes, options))
    {
        for (long offset = -1; offset >= -fileSize; offset--)
        {
            fileStream.Seek(offset, SeekOrigin.End);
        }
    }
}
```

Benchmark results

Method	Runtime	options	Windows Ratio	Unix Ratio
SeekForward	.NET 5.0	None	1.00	1.00
SeekForward	.NET 6.0	None	**0.10**	**0.04**
SeekBackward	.NET 5.0	None	1.00	1.00
SeekBackward	.NET 6.0	None	**0.03**	1.00
SeekForward	.NET 5.0	Asynchronous	1.00	1.00
SeekForward	.NET 6.0	Asynchronous	**0.02**	**0.04**
SeekBackward	.NET 5.0	Asynchronous	1.00	1.00
SeekBackward	.NET 6.0	Asynchronous	**0.01**	1.00

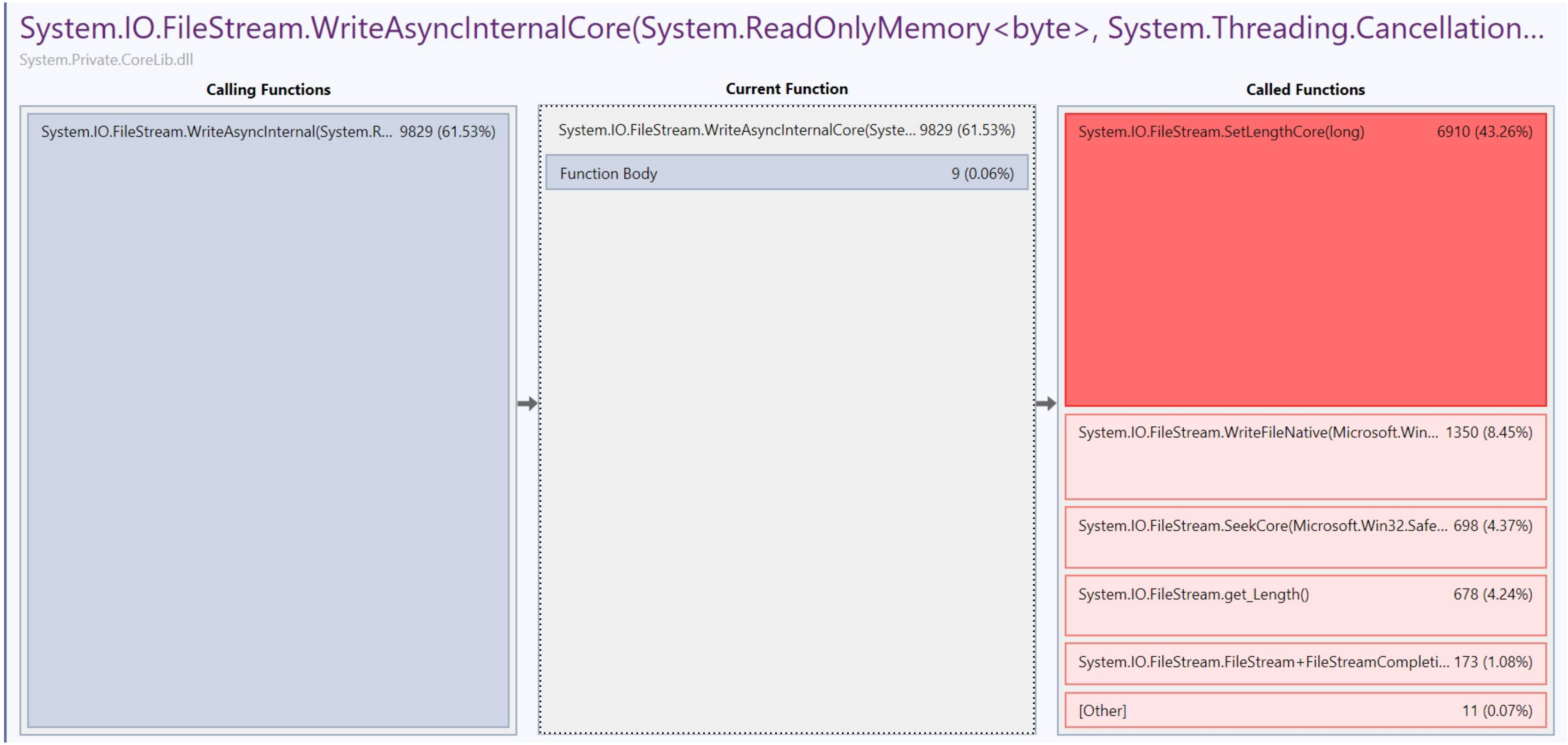
Why is there no perf improvement for SeekBackward on Unix?

Length #49975

- As long as file is not shared for writing (FileShare.Write) nobody else is able to modify the file.
- Is it true for both Windows and Unix?
- The first-time access has not changed.
- Every next call to FileStream.Length can be few orders of magnitude faster.

Method	Runtime	share	Mean	Ratio
GetLength	.NET 5.0	Read	1,932.00 ns	1.00
GetLength	.NET 6.0	Read	58.52 ns	**0.03**

WriteAsync: Windows investigation



WriteAsync old implementation pseudocode

```
long _position;
SafeFileHandle _handle;

async ValueTask WriteAsyncBefore(ReadOnlyMemory<byte> buffer)
{
  long oldEndOfFile = GetFileLength(_handle); // 1st sys-call
  long newEndOfFile = _position + buffer.Length;

  // following was true for EVERY write to an empty file
  if (newEndOfFile > oldEndOfFile)
    ExtendTheFile(_handle, newEndOfFile); // 2nd sys-call

  SetFilePosition(_handle, newEndOfFile); // 3rd sys-call
  _position += buffer.Length;

  await WriteFile(_handle, buffer); // 4th sys-call
}
```

WriteAsync new implementation pseudocode

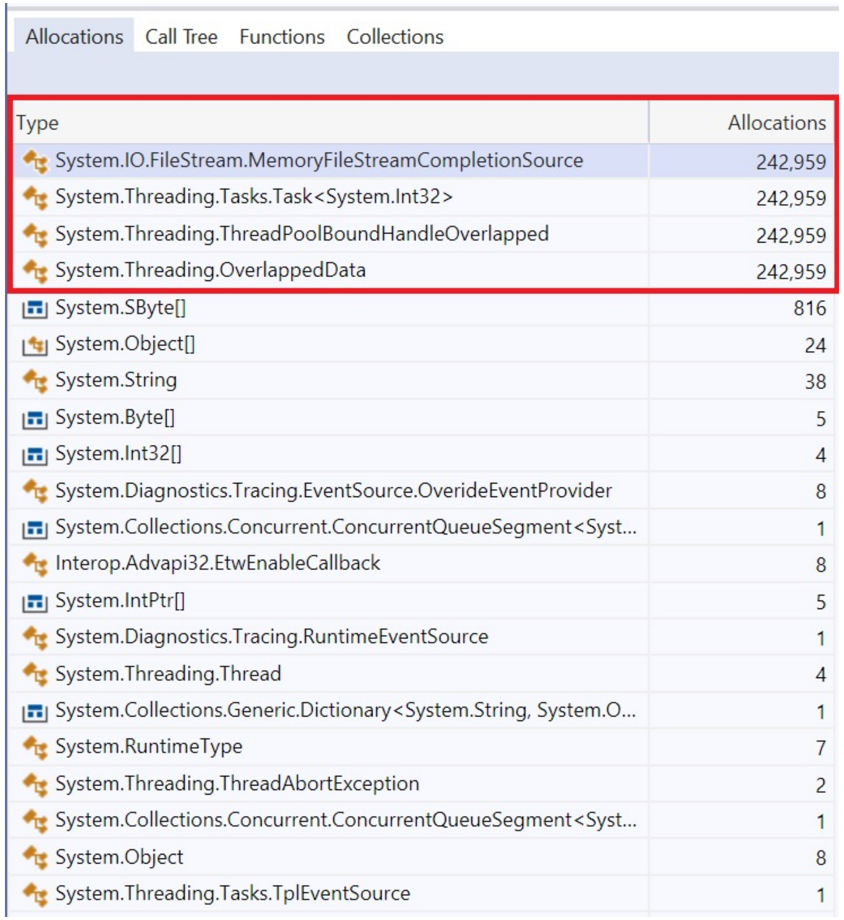
```
async ValueTask WriteAsyncAfter(ReadOnlyMemory<byte> buffer)
{
    await WriteFile(_handle, buffer, _position);
    _position += buffer.Length;
}
```

Reducing memory allocations

- #50802 TaskCompletionSource

  -> IValueTaskSource

- #51363
  - reuse IValueTaskSource instances
  - changed the ownership of OverlappedData (#25074)
  - eliminated OverlappedData and ThreadPoolBoundHandleOverlapped allocs



Allocations Call Tree Functions Collections

Type	Allocations
System.IO.FileStream.MemoryFileStreamCompletionSource	242,959
System.Threading.Tasks.Task<System.Int32>	242,959
System.Threading.ThreadPoolBoundHandleOverlapped	242,959
System.Threading.OverlappedData	242,959
System.SByte[]	816
System.Object[]	24
System.String	38
System.Byte[]	5
System.Int32[]	4
System.Diagnostics.Tracing.EventSource.OverideEventProvider	8
System.Collections.Concurrent.ConcurrentQueueSegment<Syst...	1
Interop.Advapi32.EtwEnableCallback	8
System.IntPtr[]	5
System.Diagnostics.Tracing.RuntimeEventSource	1
System.Threading.Thread	4
System.Collections.Generic.Dictionary<System.String, System.O...	1
System.RuntimeType	7
System.Threading.ThreadAbortException	2
System.Collections.Concurrent.ConcurrentQueueSegment<Syst...	1
System.Object	8
System.Threading.Tasks.TplEventSource	1

WriteAsync: benchmark

```csharp
[Benchmark]
[ArgumentsSource(nameof(AsyncArguments))]
public Task WriteAsync(long fileSize, int userBufferSize, FileOptions options)
    => WriteAsync(fileSize, userBufferSize, options, streamBufferSize: FourKibibytes);

[Benchmark]
[ArgumentsSource(nameof(AsyncArguments_NoBuffering))]
public Task WriteAsync_NoBuffering(long fileSize, int userBufferSize, FileOptions options)
    => WriteAsync(fileSize, userBufferSize, options, streamBufferSize: 1);

async Task WriteAsync(long fileSize, int userBufferSize, FileOptions options, int streamBufferSize)
{
    CancellationToken cancellationToken = CancellationToken.None;
    Memory<byte> userBuffer = new Memory<byte>(_userBuffers[userBufferSize]);

    using (FileStream fs = new FileStream(_destinationFilePaths[fileSize], FileMode.Create, FileAccess.Write,
        FileShare.Read, streamBufferSize, options))
    {
        for (int i = 0; i < fileSize / userBufferSize; i++)
        {
            await fs.WriteAsync(userBuffer, cancellationToken);
        }
    }
}
```

WriteAsync: .NET 5 vs 6 on Windows

Method	Runtime	fileSize	userBufferSize	Mean	Ratio	Allocated
WriteAsync	.NET 5.0	1024	1024	433.01 µs	1.00	**4,650 B**
WriteAsync	.NET 6.0	1024	1024	402.73 µs	**0.93**	**4,689 B**

WriteAsync: .NET 5 vs 6 on Windows

Method	Runtime	fileSize	userBufferSize	Mean	Ratio	Allocated
WriteAsync	.NET 5.0	1024	1024	433.01 μs	1.00	4,650 B
WriteAsync	.NET 6.0	1024	1024	402.73 μs	0.93	4,689 B
WriteAsync	.NET 5.0	1048576	512	9,140.81 μs	1.00	**41,608 B**
WriteAsync	.NET 6.0	1048576	512	5,762.94 μs	**0.63**	**5,425 B**

WriteAsync: .NET 5 vs 6 on Windows

Method	Runtime	fileSize	userBufferSize	Mean	Ratio	Allocated
WriteAsync	.NET 5.0	1024	1024	433.01 μs	1.00	4,650 B
WriteAsync	.NET 6.0	1024	1024	402.73 μs	0.93	4,689 B
WriteAsync	.NET 5.0	1048576	512	9,140.81 μs	1.00	41,608 B
WriteAsync	.NET 6.0	1048576	512	5,762.94 μs	0.63	5,425 B
WriteAsync	.NET 5.0	1048576	4096	21,214.05 μs	1.00	**80,320 B**
WriteAsync	.NET 6.0	1048576	4096	4,711.63 μs	**0.22**	**940 B**

WriteAsync: .NET 5 vs 6 on Windows

Method	Runtime	fileSize	userBufferSize	Mean	Ratio	Allocated
WriteAsync	.NET 5.0	104857600	**4096**	**2.61 s**	1.00	**7,987,648 B**
WriteAsync	.NET 6.0	104857600	**4096**	**0.42 s**	0.16	**2,272 B**
WriteAsync_NoBuffering	.NET 5.0	104857600	**16384**	**0.77 s**	1.00	**1,997,248 B**
WriteAsync_NoBuffering	.NET 6.0	104857600	**16384**	**0.14 s**	0.19	**1,832 B**

Perf tip #5: user buffer size has great perf impact

```
public class UserBuffer
{
    private const string FilePath = "file.data";
    private const int FileSize = 100_000_000; // 100 MB

    [Params(1_000, 4_000, 8_000, 16_000, 32_000, 64_000)]
    public int UserBufferSize;

    private byte[] _userBuffer;

    [GlobalSetup]
    public void Setup()
    {
        _userBuffer = new byte[UserBufferSize];
        File.WriteAllBytes(FilePath, new byte[FileSize]);
    }

    [Benchmark]
    public void Read()
    {
        using (FileStream fs = File.OpenRead(FilePath))
        while (fs.Read(_userBuffer) != 0) ;
    }

    [Benchmark]
    public void Write()
    {
        using (FileStream fs = File.OpenWrite(FilePath))
        for (int i = 0; i < FileSize / UserBufferSize; i++)
            fs.Write(_userBuffer);
    }
}
```

Method	UserBufferSize	Mean
Read	**1000**	**86.29 ms**
Read	4000	61.77 ms
Read	8000	38.32 ms
Read	16000	27.62 ms
Read	32000	21.72 ms
Read	**64000**	**18.68 ms**

Method	UserBufferSize	Mean
Write	**1000**	**69.18 ms**
Write	4000	68.43 ms
Write	8000	55.31 ms
Write	16000	43.29 ms
Write	32000	36.91 ms
Write	**64000**	**33.25 ms**

WriteAsync: Unix implementation

- It was already performing 1 sys-call per WriteAsync.
- #55123 has combined the concept of IValueTaskSource and IThreadPoolWorkItem into a single type
- By implementing IThreadPoolWorkItem interface, the type gained the possibility of queueing itself on the Thread Pool (which normally requires an allocation of a ThreadPoolWorkItem)
- By re-using IValueTaskSource instances, achieved amortized allocation-free file operations.

WriteAsync: Unix improvements

Method	Runtime	fileSize	userBufferSize	options	Mean	Ratio	Allocated
WriteAsync	.NET 5.0	1024	1024	None	53.002 μs	1.00	4,728 B
WriteAsync	.NET 6.0	1024	1024	None	34.615 μs	**0.65**	4,424 B
WriteAsync	.NET 5.0	1024	1024	Asynchronous	34.020 μs	1.00	4,400 B
WriteAsync	.NET 6.0	1024	1024	Asynchronous	33.699 μs	0.99	4,424 B

WriteAsync: Unix improvements

Method	Runtime	fileSize	userBufferSize	options	Mean	Ratio	Allocated
WriteAsync	.NET 5.0	1048576	512	None	5,531.106 μs	1.00	**234,004 B**
WriteAsync	.NET 6.0	1048576	512	None	2,133.012 μs	**0.39**	**5,002 B**
WriteAsync	.NET 5.0	1048576	512	Asynchronous	2,447.687 μs	1.00	**33,211 B**
WriteAsync	.NET 6.0	1048576	512	Asynchronous	2,121.449 μs	**0.87**	**5,009 B**

- In this particular benchmark we take advantage of buffering (userBufferSize = 512, bufferSize = 4096)
- In #56095 we have started to pool the async method builder by just applying PoolingAsyncValueTaskMethodBuilder attributes:

```
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))]
async ValueTask<int> ReadAsync(…)

[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder))]
async ValueTask WriteAsync(..)
```

WriteAsync: Unix improvements

Method	Runtime	fileSize	userBufferSize	options	Mean	Ratio	Allocated
WriteAsync	.NET 5.0	1048576	4096	None	2,296.017 μs	1.00	**29,170 B**
WriteAsync	.NET 6.0	1048576	4096	None	1,889.585 μs	**0.83**	**712 B**
WriteAsync	.NET 5.0	1048576	4096	Asynchronous	2,024.704 μs	1.00	18,986 B
WriteAsync	.NET 6.0	1048576	4096	Asynchronous	1,897.600 μs	**0.94**	**712 B**

ReadAsync old implementation pseudocode

```
async ValueTask<int> ReadAsyncBefore(Memory<byte> buffer)
{
   long fileOffset = _position;
   long endOfFile = GetFileLength(_handle);

   if (fileOffset + buffer.Length > endOfFile) // read beyond EOF
     buffer = buffer.Slice(0, endOfFile - fileOffset);

   _position = SetFilePosition(_handle, fileOffset + buffer.Length);

   await ReadFile(_handle, buffer);
}
```

ReadAsync new implementation pseudocode

```
async ValueTask<int> ReadAsyncAfter(Memory<byte> buffer)
{
  int bytesRead = await ReadFile(_handle, buffer, _position);
  _position += bytesRead;

  return bytesRead;
}
```

ReadAsync: benchmarks

```
[Benchmark]
[ArgumentsSource(nameof(AsyncArguments))]
public Task<long> ReadAsync(long fileSize, int userBufferSize, FileOptions options)
    => ReadAsync(fileSize, userBufferSize, options, streamBufferSize: FourKibibytes);

[Benchmark]
[ArgumentsSource(nameof(AsyncArguments_NoBuffering))]
public Task<long> ReadAsync_NoBuffering(long fileSize, int userBufferSize, FileOptions options)
    => ReadAsync(fileSize, userBufferSize, options, streamBufferSize: 1);

async Task<long> ReadAsync(long fileSize, int userBufferSize, FileOptions options, int streamBufferSize)
{
    CancellationToken cancellationToken = CancellationToken.None;
    Memory<byte> userBuffer = new Memory<byte>(_userBuffers[userBufferSize]);
    long bytesRead = 0;
    using (FileStream fileStream = new FileStream(
        _sourceFilePaths[fileSize], FileMode.Open, FileAccess.Read, FileShare.Read, streamBufferSize, options))
    {
        while (bytesRead < fileSize)
        {
            bytesRead += await fileStream.ReadAsync(userBuffer, cancellationToken);
        }
    }

    return bytesRead;
}
```

ReadAsync: Windows benchmark results

Method	Runtime	fileSize	userBufferSize	options	Mean	Ratio	Allocated
ReadAsync	.NET 5.0	1048576	512	Asynchronous	5,163.71 μs	1.00	41,479 B
ReadAsync	.NET 6.0	1048576	512	Asynchronous	3,406.73 μs	**0.66**	**5,233 B**
ReadAsync	.NET 5.0	1048576	4096	Asynchronous	6,575.26 μs	1.00	80,320 B
ReadAsync	.NET 6.0	1048576	4096	Asynchronous	2,873.59 μs	**0.44**	**936 B**
ReadAsync_NoBuffering	.NET 5.0	1048576	16384	Asynchronous	1,915.17 μs	1.00	20,420 B
ReadAsync_NoBuffering	.NET 6.0	1048576	16384	Asynchronous	856.61 μs	**0.45**	**782 B**
ReadAsync	.NET 5.0	104857600	4096	Asynchronous	714,699.30 μs	1.00	7,987,648 B
ReadAsync	.NET 6.0	104857600	4096	Asynchronous	297,675.86 μs	**0.42**	**2,272 B**
ReadAsync_NoBuffering	.NET 5.0	104857600	16384	Asynchronous	192,485.40 μs	1.00	1,997,248 B
ReadAsync_NoBuffering	.NET 6.0	104857600	16384	Asynchronous	93,350.07 μs	**0.49**	**1,040 B**

ReadAsync: Unix benchmark results

Method	Runtime	fileSize	userBufferSize	options	Mean	Ratio	Allocated
ReadAsync	.NET 5.0	1048576	512	None	3,550.898 μs	1.00	233,997 B
ReadAsync	.NET 6.0	1048576	512	None	674.037 μs	**0.19**	**5,019 B**
ReadAsync	.NET 5.0	1048576	512	Asynchronous	744.525 μs	1.00	35,369 B
ReadAsync	.NET 6.0	1048576	512	Asynchronous	663.037 μs	**0.91**	**5,019 B**
ReadAsync	.NET 5.0	1048576	4096	None	537.004 μs	1.00	29,169 B
ReadAsync	.NET 6.0	1048576	4096	None	375.843 μs	**0.72**	**706 B**
ReadAsync	.NET 5.0	1048576	4096	Asynchronous	499.676 μs	1.00	31,249 B
ReadAsync	.NET 6.0	1048576	4096	Asynchronous	398.217 μs	**0.81**	**706 B**
ReadAsync_NoBuffering	.NET 5.0	1048576	16384	None	187.578 μs	1.00	7,664 B
ReadAsync_NoBuffering	.NET 6.0	1048576	16384	None	154.951 μs	**0.83**	**553 B**
ReadAsync_NoBuffering	.NET 5.0	1048576	16384	Asynchronous	189.687 μs	1.00	8,208 B
ReadAsync_NoBuffering	.NET 6.0	1048576	16384	Asynchronous	158.541 μs	**0.84**	**553 B**
ReadAsync	.NET 5.0	104857600	4096	None	49,196.600 μs	1.00	2,867,768 B
ReadAsync	.NET 6.0	104857600	4096	None	41,890.758 μs	**0.85**	**1,124 B**
ReadAsync	.NET 5.0	104857600	4096	Asynchronous	48,793.215 μs	1.00	3,072,600 B
ReadAsync	.NET 6.0	104857600	4096	Asynchronous	42,725.572 μs	**0.88**	**1,124 B**
ReadAsync_NoBuffering	.NET 5.0	104857600	16384	None	23,819.030 μs	1.00	717,354 B
ReadAsync_NoBuffering	.NET 6.0	104857600	16384	None	18,961.480 μs	**0.80**	**644 B**
ReadAsync_NoBuffering	.NET 5.0	104857600	16384	Asynchronous	21,595.085 μs	1.00	768,557 B
ReadAsync_NoBuffering	.NET 6.0	104857600	16384	Asynchronous	18,861.580 μs	**0.87**	**668 B**

Summary: .NET 6 Performance improvements

- File position is no longer synced with the OS, it’s stored in memory.
- Seeking forward is few dozens times faster.
- Seeking backward is faster on Windows, but not on Unix.
- File Length for !FileShare.Write is now cached on Windows.
- WriteAsync is up to five times faster on Windows.
- ReadAsync is up to two times faster on Windows.
- Async operations are on average 20% faster on Unix.
- All async file operations are allocation free (except the first call).

New APIs: Thread-safe file IO #53669

```
namespace System.IO
{
  public static class RandomAccess
  {
    public static int Read(SafeFileHandle handle, Span<byte> buffer, long fileOffset);

    public static void Write(SafeFileHandle handle, ReadOnlySpan<byte> buffer, long fileOffset);

    public static ValueTask<int> ReadAsync(SafeFileHandle handle,
      Memory<byte> buffer, long fileOffset, CancellationToken cancellationToken = default);

    public static ValueTask WriteAsync(SafeFileHandle handle, ReadOnlyMemory<byte> buffer,
      long fileOffset, CancellationToken cancellationToken = default);

    public static long GetLength(SafeFileHandle handle);
  }
}
```

Sample: writing to file

```
async Task ThreadSafeAsync(string path,
    IReadOnlyList<ReadOnlyMemory<byte>> buffers)
{
    using SafeFileHandle handle = File.OpenHandle(path,
        FileMode.Create, FileAccess.Write,
        FileShare.None, FileOptions.Asynchronous);

    long offset = 0;
    for (int i = 0; i < buffers.Count; i++)
    {
        await RandomAccess.WriteAsync(handle, buffers[i], offset);
        offset += buffers[i].Length;
    }
}
```

New APIs: Scatter/Gather IO

```
namespace System.IO
{
  public static class RandomAccess
  {
    public static long Read(SafeFileHandle handle,
      IReadOnlyList<Memory<byte>> buffers, long fileOffset);
    public static void Write(SafeFileHandle handle,
      IReadOnlyList<ReadOnlyMemory<byte>> buffers, long fileOffset);

    public static ValueTask<long> ReadAsync(SafeFileHandle handle,
      IReadOnlyList<Memory<byte>> buffers, long fileOffset, CancellationToken cancellationToken = default);
    public static ValueTask WriteAsync(SafeFileHandle handle,
      IReadOnlyList<ReadOnlyMemory<byte>> buffers, long fileOffset, CancellationToken cancellationToken = default);
  }
}
```

Sample: writing to file

```
async Task OptimalSysCallsAsync(string path,
    IReadOnlyList<ReadOnlyMemory<byte>> buffers)
{
    using SafeFileHandle handle = File.OpenHandle(
        path, FileMode.Create,
        FileAccess.Write, FileShare.None, FileOptions.Asynchronous);

    await RandomAccess.WriteAsync(handle, buffers, fileOffset: 0);
}
```

Perf tip #6: Reduce the number of sys-calls

- You can use the new APIs to reduce the number of sys-calls
- It works for every Linux (gains up to 10%), newer macOSes
- Windows has very strict requirements:
  - async file handles
  - NO_BUFFERING
  - All buffers aligned and of size equal Environment.SystemPageSize
- If requirements are not meet, it performs buffer-many sys-calls.

New APIs: FileStreamOptions

```
namespace System.IO
{
    public sealed class FileStreamOptions
    {
        public FileStreamOptions() {}
        public FileMode Mode { get; set; }
        public FileAccess Access { get; set; } = FileAccess.Read;
        public FileShare Share { get; set; } = FileShare.Read;
        public FileOptions Options { get; set; }
        public int BufferSize { get; set; } = 4096;
        public long PreallocationSize { get; set; }
    }

    public class FileStream : Stream
    {
        public FileStream(string path, FileStreamOptions options);
    }
}
```

Sample: FileStreamOptions

```
var openForReading = new FileStreamOptions { Mode = FileMode.Open };
using FileStream source = new FileStream("source.txt", openForReading);

var createForWriting = new FileStreamOptions
{
    Mode = FileMode.CreateNew,
    Access = FileAccess.Write,
    BufferSize = 0, // disable FileStream buffering
    PreallocationSize = source.Length // specify size up-front
};
using FileStream destination = new FileStream("destination.txt", createForWriting);
source.CopyTo(destination);
```

Perf tip #7: specify size up-front

- PreallocationSize is a hint, not a strong guarantee.
- It’s supported on Windows, Linux and macOS. Not on WASM or BSD.
- When there is not enough space or the file is too large, it throws.
- It supports FileMode.Create and FileMode.CreateNew.
- It pre-allocates disk size but does not modify EOF.
- It improves perf:
  - Write operations don’t need to extend the file
  - It’s less likely for the file to be fragmented
- For File Systems where modyfing EOF is expensive like ext4 or for older runtimes, you may use FileStream.SetLength() to set EOF.

Summary: new APIs

- Opening file handles: File.OpenHandle
- Thread-safe file IO: RandomAccess
- Scatter/Gather IO: RandomAccess
- PreallocationSize
- FileStreamOptions

Performance Guidelines: benchmarking

- How the code is going to be used in production?
  - What is the target OS?
  - Is disk encryption going to be enabled?
  - What is the target File System?
  - What are the file sizes?
  - What is the hardware? SSD?
- Measure, measure, measure!

Performance Guidelines: Opening Files

- Use async file IO if you need scalability or cancellation support.
- Don’t mix sync and async.
- Do you need buffering? If not, disable it.
- Do you know the size up-front? If so, specify PreallocationSize.
- If you use FileShare.Write, Length won’t be cached.
- Prefer File.OpenHandle over FileStream.SafeFileHandle.

Performance Guidelines: fewer sys-calls

- Use large buffers to reduce the number of sys-calls.
- You can pool the buffers using ArrayPool to avoid allocations.
- You can allocate aligned buffers using NativeMemory.AlignedAlloc.
- You can reduce the number of sys-calls even further by using Scatter/Gather APIs.

Questions?

Thank you!

@SitnikAdam

Adam.Sitnik@microsoft.com