Distributed Mutual Exclusion using Proposed Boost.AFIO
(asynchronous filesystem and file io)

Niall Douglas

Contents:

1. Quick overview of proposed Boost.AFIO and its current status: v1 and v2
2. Overview of API design of Abstract Base Class `afio`::`algorithm`:: `shared_fs_mutex`
3. Recap from CppCon tutorial: The four base techniques of concurrency control on the file system

Contents:

4. High-level overview of our atomic append lock algorithm
5. Detail of the atomic append lock algorithm implementation
6. Benchmarks comparing our atomic append lock algorithm implementation to lock files and byte range locks

History and status of proposed Boost.AFIO

Proposed Boost.AFIO v1 (2012-2015)

Proposed Boost.AFIO v1

- Provides a single universal file system programming model across all OSs
  - Fully featured on Windows and Linux
  - Reduced featured on FreeBSD and OS X
- Where a platform is deficient in host OS support, where possible a feature is emulated, even if quite inefficiently
  - Raw performance is secondary to correctness and cross-platform consistency of behaviours

Proposed Boost.AFIO - current status

- Entered Boost peer review queue in October 2013
- Was peer reviewed by Boost community August 2015
  - Universal rejection by all reviewers bar one
- Biggest problem areas noted by peer reviewers were:
  - Too heavyweight e.g. mandatory use of reference counting for lifetime management, use of exceptions for error transport, mandatory use of thread pool
  - Lack of breakout of internal routines into a public API people can mash up on their own
  - Didn't solve any problem people think they have/Documentation was confusing
  - Performance was lower than using the host OS APIs directly

Proposed Boost.AFIO v2 (2015-)

Proposed Boost.AFIO v2

- "Bare metal" design
  - Exposes all the quirks of the host OS to user unfiltered
  - No threads, no resource nor memory management, no exceptions (we return lightweight monadic *Outcomes* instead)
  - Performance never measurably worse than using host OS APIs directly - overhead often just *hundreds* of assembler opcodes
- Tight mapping between C++ type system and filing system object primitives
  - Made possible by dropping WinXP support
- Publicly exposes core file system algorithms library into afio::algorithm::*

Proposed Boost.AFIO v2 - status

- Very early alpha code - use v1 for now!
- No test suite yet - at all!
- Currently only works on Microsoft Windows
  - Getting POSIX to do reliable delete-on-last-close semantics was much harder than expected (sorry!)
- v2 dependencies are >= VS2015 C++ 14, Core C++ GSL (Martin's gsl-lite), Filesystem TS
  - Designed around coming C++ 1z features (borrowing spans, auto free functions, contiguity guarantees, custom attributes, STL v2)

afio::algorithm::shared_fs_mutex

There are these implementations of shared_fs_mutex in AFIO v2:

1. shared_fs_mutex::lock_files
2. shared_fs_mutex::byte_ranges
3. shared_fs_mutex::atomic_append
   - (the one we are about to workshop now)
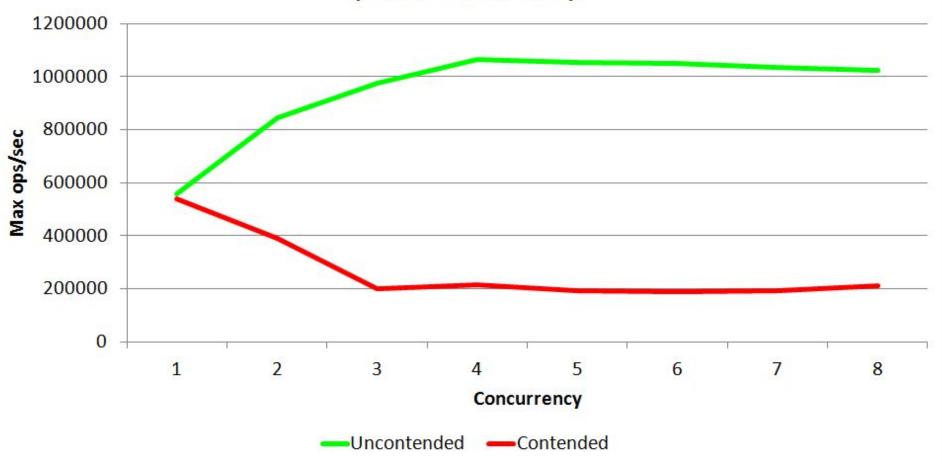
All implement the abstract base class shared_fs_mutex::shared_fs_mutex

shared_fs_mutex abstract base class

Its API design is due to:

- Modern filing systems tend to parallelise quite well
  - For uncontended locks on the same object, typically aggregate performance improves up to CPU count, after which plateaus before beginning a linear slow decline
  - For contended locks, constant time on the OS process context switch time

Scaling of byte range locks to concurrency on NTFS
(2 core 4 thread CPU)
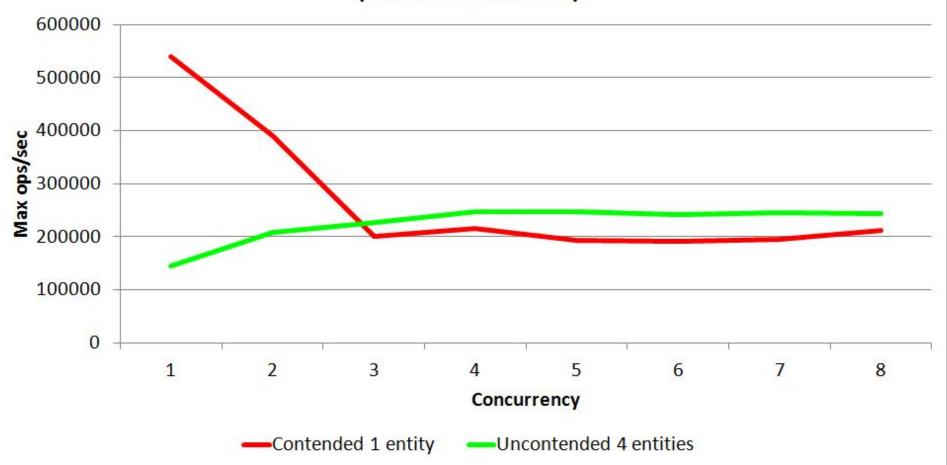
shared_fs_mutex abstract base class

- It therefore usually improves overall performance to push granularity of locking onto the file system

- So, instead of giant single locks which contend more frequently, use many fine grained locks which contend **rarely**

Scaling of byte range locks to concurrency on NTFS
(2 core 4 thread CPU)

shared_fs_mutex abstract base class

- virtual shared_fs_mutex::lock() therefore takes a span<> of multiple *entities* to lock
  - An entity is simply a 63 bit number
  - Top bit set means <u>exclusive</u>, unset means <u>shared</u>
    - (note top bit is useless for byte range locks on POSIX, and we internally use the top bit on Windows to emulate advisory locks)
- To entity conversion functions are provided:
  - Crypto random, from string, from span<T> etc

afio::algorithm::shared_fs_mutex

- How entities are mapped onto the file system depends on the implementation, so:
  - shared_fs_mutex::lock_file maps entities into 16 character hexadecimal files opened with O_EXCL
  - shared_fs_mutex::byte_ranges maps entities into single byte offsets into the shared lock file
- Contention is handled by backing off all preceding locks and randomising the list before trying again, starting with the entity which was contended last try

So why do we need shared_fs_mutex::atomic_append?

- A problem with lock files/byte range locks is the inverse log scaling to entity count
- It gets much worse with high waiters and high entities on a contended lock
  - *Hyperbolic* drop off in performance after a certain system dependent point - a "write hole"
- What we need is an algorithm with an <u>excellent worst case</u>: hello atomic_append!

atomic_append

- Before we can design a correct atomic append algorithm, we need to understand Concurrency Control primitives on the file system in order to glue together an efficient and correct implementation

(for an expansion to the following theory overview, see CppCon 2015 tutorial "Racing the filesystem" https://www.youtube.com/watch?v=uhRWMGBjlO8)

Further reading:

AFIO v2 API reference:
https://ned14.github.io/boost.afio

Abstract base class
algorithm::shared_fs_mutex API reference:
https://goo.gl/NG9ttv

Concurrency control on the file system

A very rapid theory primer

File system concurrency control:

- Needed anywhere where more than one process may modify storage [potentially] concurrent to other processes
- POSIX specifies strong read/write reordering visibility guarantees (atomic, acquire/release) but implementations vary **extensively** in conformance

File system concurrency primitives:

There are four types of concurrency control possible on the file system (in order of increasing performance):

1. Exclusive lock files (easiest, most portable)
2. Byte range locks (easy on Windows and recent Linux, dragons live on non-Linux POSIX, tricky on SMB, NFS)
3. Atomic append + extent deallocation (extent based i.e. recent filing systems only)
4. POSIX change visibility ordering guarantees (file system geeks only, dragons abound here especially on Windows)

1. Exclusive lock files

Exclusive lock files

Pros:

- Works as expected between operating systems on the same networked drive

Cons:

- Exclusive only i.e. cannot permit multiple readers
- Not sudden power loss friendly
- On POSIX, race free breaking stale lock files from unexpected process exit is a problem
  - Windows has very useful delete-on-last-close facility
- No way of efficiently sleeping until a lock file is freed
  - Expensive on CPU and battery, bad for mobile devices

2. Byte Range Locks

Byte range locks

Pros:

- Allows non-modifying operations to parallelise
- Automatically unlocks on sudden process exit
- No problems with unexpected power off
- Thread can be slept waiting for lock (blocking)

Cons:

- Limited viable use cases (inode based!) on POSIX except Linux >= 3.15
  - On POSIX any single close() unlocks all locks for that inode for all fds in the process ☹
- Problematic on shared networked drives
  - Advisory on POSIX, mandatory on Windows, plus on POSIX offset and length is signed unfortunately

3. Atomic append + Extent deallocation

(we'll be making use of this for our algorithm)

Atomic append + extent deallocation

- On everything including SMB except NFS, writes to append-only files are *atomic*
  - i.e. concurrent writes are never interleaved *in append offset chosen*
- Extent-based filing systems allow arbitrary deallocation of ranges of a [sparse] file
  - i.e. they no longer consume physical storage

Atomic append + extent deallocation

- Combining these facilities allows safe concurrent file updates through appending whatever the change is and deallocating any obsoleted data
  - File grows "forever" but actually doesn't
  - If deallocation done frequently, kernel page cache will usually skip writing any actual data to disk, so in practice this is a pure memory IPC technique highly dependent on memory bandwidth

Atomic append + extent deallocation

Pros:

- Very fast EXCEPT on ZFS
  - (I need to submit a bug report actually …)
- Works without quirks on all configurations except NFS
- Only portable way of achieving *late durability*
- With a bit of mind warping, technique is surprisingly algorithmically flexible e.g. a distributed mutual exclusion algorithm (Suzuki & Kasami; Maekawa & Ricart; Agrawala)

Atomic append + extent deallocation

Cons:

- Requires an extents-based filing system (anything created in the past 30 years is usually extents-based) otherwise file grows in physical storage forever
  - One can use *segmented files* (slow, i/o intensive!) to work around this on old filing systems
- Performs best if appended records are "chunky"
  - Extent granularity is anywhere between 4Kb and 128Kb depending on filing system

4. POSIX concurrent change visibility ordering guarantees

(brutally incomplete summary of just enough detail for this workshop)

Atomicity of writes w.r.t. reads

Empirical test with kernel page cache enabled
i.e.
O_DIRECT=off/FILE_FLAG_NO_BUFFERING=off

Microsoft Windows 10	NTFS	1 byte
Linux 4.2.10	ext4	1 byte
FreeBSD 10.2	ZFS	>= 1Mb

Atomicity of writes w.r.t. reads

Kernel page cache disabled i.e.
O_DIRECT=on/FILE_FLAG_NO_BUFFERING=on

Microsoft Windows 10	NTFS	>= cache line (64 bytes) <= PCIe burst (4096)
Linux 4.2.10	ext4	>= 1Mb
FreeBSD 10.2	ZFS	>= 1Mb

The atomic append lock algorithm - Overview

Questions before we begin this section?

Atomic append lock algorithm overview

- To be honest I started with a modified Suzuki-Kasami algorithm
  - Hence this talk's description during Call for Papers
- But as I kept optimising the implementation I arrived at something quite different
- Only last weekend did I Google to see if someone had invented my algorithm already - the answer is "sorta" …

Eisenburg & McGuire algorithm

Algorithm from their 1972 paper:

https://en.wikipedia.org/wiki/Eisenberg_%26_McGuire_algorithm

Key characteristics for N processes:

- Array[N] of IDLE, WAITING, ACTIVE states
- On begin to lock:
  - Set my state to WAITING
  - Busy spin from me to the lock holder until all intervening states are IDLE
  - Set my state to ACTIVE
  - First ACTIVE state gets the lock

Lamport's Bakery algorithm

Algorithm from his 1974 paper:

https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm

Key characteristics for N processes:

- Each new entrant receives a unique number from a "Lamport clock"
- We loop scanning the shared list of other waiters until our number is the lowest
- Can be easily made FIFO and therefore bounded wait and starvation free

Peterson's N-process algorithm

Algorithm from his 1981 paper:

https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Key characteristics for N processes:

- List of (N-1) "waiting rooms" shared by all users
- Processes advance from Room[0] to Room[N-1], being reduced by one with each advance
- Room[N-1] is the critical section
- Starvation free
- But no bounded waiting!

The atomic_append algorithm

Ours is a mix of all the above:

- The atomically appended file is the shared FIFO queue of waiters
- The act of atomic append is the "Lamport clock" via an always incrementing file length where lower offsets are relatively closer to gaining the lock
- Therefore there is a total sequentially consistent ordering from the end of the file towards the beginning

The atomic_append algorithm

To lock:

1. Atomic append our lock request for up to M entities
2. Scan from our lock request backwards looking for other requests locking same thing as us. If found, spin reading them until they become all bits zero
3. When we reach the front of the list, we have the lock!

To unlock:

1. Fill our lock request with all bits zero

The atomic_append algorithm

Characteristics:

- Strict FIFO fairness
- Guaranteed progress
- Ideal execution complexity

I believe this is actually an _ideal_ mutual exclusion algorithm!

The atomic append lock algorithm - Detail

Questions before we begin this section?

```cpp
#pragma pack(1)
    struct alignas(16) header
    {
      uint128 hash;                      // Hash of remaining 112 bytes
      uint64 generation;                 // Iterated per write
      uint64 time_offset;                // time_t in seconds at time of creation. Used to offset us_coun
      uint64 first_known_good;           // offset to first known good lock_request
      uint64 first_after_hole_punch;     // offset to first byte after last hole punch
      // First 48 bytes are the header, remainder is zeros for future expansion
      uint64 _padding[10];
      // Last byte is used to detect first user of the file
    };
    static_assert(sizeof(header) == 128, "header structure is not 128 bytes long!");

    struct alignas(16) lock_request
    {
      uint128 hash;                                    // Hash of remaining 112 bytes
      uint64 unique_id;                                // A unique id identifying this locking instance
      uint64 us_count : 56;                            // Microseconds since the lock file created
      uint64 items : 8;                                // The number of entities below which are valid
      shared_fs_mutex::entity_type entities[12];       // Entities to exclusive or share lock
    };
    static_assert(sizeof(lock_request) == 128, "lock_request structure is not 128 bytes long!");
#pragma pack(pop)
```

Questions?

atomic_append::init()

1. Open the shared lock file with read/write access and delete-on-last-close semantics
2. Try to lock for exclusive access the final byte in the header
   a. If successful, truncate file and write new header. Atomic downgrade exclusive lock to shared
   b. If unsuccessful, lock for shared access the final byte in the header

atomic_append::init()

3. Invent a random 64 bit unique id so our lock requests can be disambiguated from other actors locking the same thing
   ○ We use the crypto strong randomness source
4. Precache the current header

   ○ On filing systems which present torn writes to readers, need to iterate reading header until 128 bit hash matches contents

Questions?

atomic_append::lock()

1. Flip the handle to the shared lock file into atomic-append-only mode
2. Record the current length of the file
3. (If NFS compatibility is needed, byte range lock file from current length to max during append)
4. Atomic append our lock_request
5. Flip the handle back to read/write mode
6. Read all bytes from previously recorded length of file to end of file in 4Kb chunks

atomic_append::lock()

7. Find the offset of the lock_request we previously atomic appended
8. Take an exclusive advisory byte range lock on our lock_request
9. Scan backwards from that offset examining each preceding lock_request until header. first_known_good reached, reloading header after if we ever contend
   a. (If hash doesn't match contents, reload)

atomic_append::lock()

b. If it is all bits zero (released), skip
c. If the entities it will lock are none of ours, skip
d. If the entities it will lock match some of ours but both are for shared access, skip
e. If the entities it will lock match some of ours but either is for exclusive access, then take a shared byte range lock on this lock_request
   - This blocks until the lock_request changes
   - Loop checking this lock_request

10. We have the lock, so return

Questions?

atomic_append::unlock()

1. Write an all bits zero lock_request to my lock_request offset
   - (some filing systems eliminate physical storage for contiguous chunks of all bits zero)
2. Release the exclusive byte range lock on my lock_request
3. If my offset was on a 4Kb boundary (i.e. every 4096/128 = 32 records) ...

atomic_append::unlock()

4. Load 6Kb of data from header.first_known_good onwards
5. Iterate until finding first non-zero record
6. If header.first_known_good - header.first_after_hole_punch >= 1Mb
   - Punch hole in physical storage
7. Write new header with updated first_known_good etc
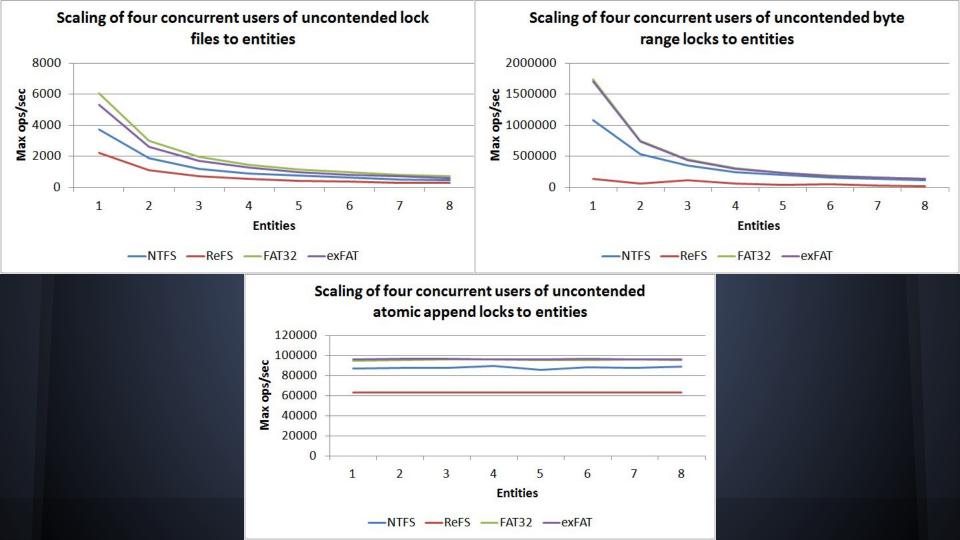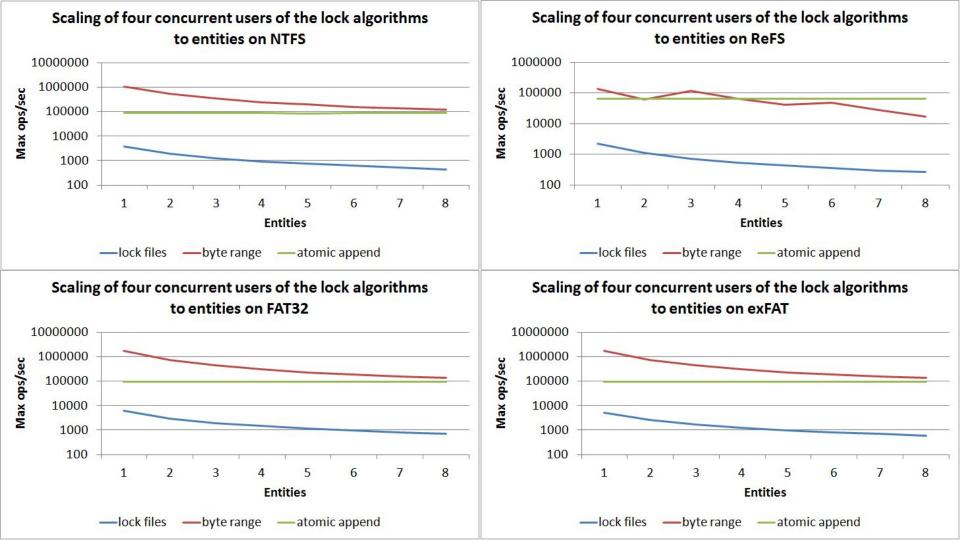
Benchmarks!
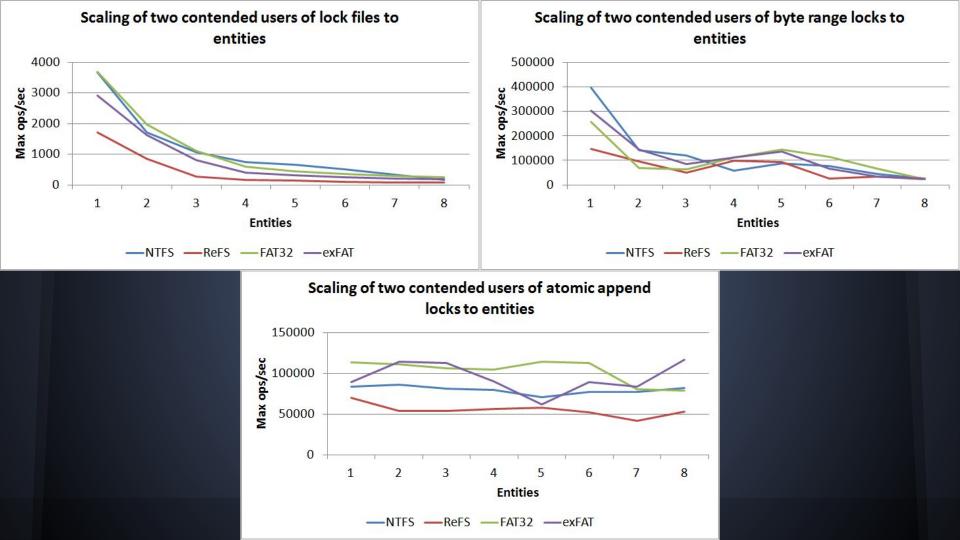
The following benchmarks are for:

2 core 4 thread 4.125Gb/sec main memory bandwidth
GenuineIntel Intel(R) Core(TM) i5 CPU M 540 @ 2.53
GHz (2008 era laptop)
Microsoft Windows 10.0.10240.16683

Known deficiencies in this atomic_append implementation:

1. Unnecessary read() between scan forward and backward for lock_request
2. Hash check failures reload too much data
3. If OS has unified page cache and SMB/NFS compatibility not important, using memory maps would be a big gain
   - Fallback to non-mapped i/o if a SMB/NFS user tries to claim lock is straightforward to add

Scaling of four concurrent users of uncontended lock files to entities

Scaling of four concurrent users of uncontended byte range locks to entities

Scaling of four concurrent users of uncontended atomic append locks to entities

Scaling of four concurrent users of the lock algorithms to entities on NTFS

Max ops/sec vs Entities

lock files — byte range — atomic append

Scaling of four concurrent users of the lock algorithms to entities on ReFS

Max ops/sec vs Entities

lock files — byte range — atomic append

Scaling of four concurrent users of the lock algorithms to entities on FAT32

Max ops/sec vs Entities

lock files — byte range — atomic append

Scaling of four concurrent users of the lock algorithms to entities on exFAT

Max ops/sec vs Entities

lock files — byte range — atomic append

Scaling of two contended users of lock files to entities

Scaling of two contended users of byte range locks to entities

Scaling of two contended users of atomic append locks to entities

NTFS

lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	3225	1548	979	773	622	522	448	404
2 waiters	3663	1717	1058	749	660	439	333	173
3 waiters	3676	1553	869	556	299	177	119	88
4 waiters	3507	1456	468	265	155	104	83	60
5 waiters	3170	1238	326	203	115	82	57	43
6 waiters	2993	958	282	161	89	62	45	31
7 waiters	2921	836	264	142	78	48	31	25
8 waiters	2771	524	221	123	62	44	25	19

byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	577811	296023	193740	149256	107958	100535	85732	75138
2 waiters	335873	142153	120077	58638	87180	76342	45294	25083
3 waiters	201341	119723	120751	111594	95092	79542	54863	28078
4 waiters	212334	113706	83377	81919	23144	32086	10601	4480
5 waiters	202346	110786	108678	97624	71422	15104	3508	2844
6 waiters	188840	93870	84231	90080	24534	4261	2642	2195
7 waiters	191017	104315	100055	68568	12964	3057	2311	2136
8 waiters	212472	103827	58576	23138	4123	2670	2249	2143

atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	62585	62207	61542	62805	62366	61434	62603	62119
2 waiters	83489	85748	81203	79872	70425	76958	77422	81651
3 waiters	74157	74859	74344	74380	74640	73540	74574	75318
4 waiters	59858	59977	60647	59355	60022	58904	60003	59328
5 waiters	57662	57868	58221	58120	57901	57932	57934	57948
6 waiters	56053	55935	56111	55512	55806	55938	55914	56223
7 waiters	55299	55422	55603	55385	55472	55213	55104	55325
8 waiters	55456	55129	55425	55370	55438	55720	55477	55741

!lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	2909	1438	367	745	585	455	440	350
2 waiters	3550	1701	1202	872	705	555	475	430
3 waiters	3834	1804	1206	910	705	589	528	441
4 waiters	3742	1884	1204	906	746	608	510	440
5 waiters	3806	1901	1267	341	753	631	533	466
6 waiters	3951	1958	1373	972	777	631	545	435
7 waiters	4046	2023	1357	1017	813	640	549	470
8 waiters	4436	2014	1402	1020	777	677	543	517

!byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	591620	291724	201182	151854	105881	101030	84476	76541
2 waiters	873678	443791	287517	213171	169548	138431	120790	105809
3 waiters	1E+06	493416	316390	227890	183447	151368	129916	114330
4 waiters	1E+06	535434	348619	245531	197438	155970	138553	118115
5 waiters	1E+06	528127	341497	250788	191613	160244	130926	115318
6 waiters	1E+06	504259	337299	250581	193143	166340	137599	115318
7 waiters	1E+06	508833	336061	246945	186329	157049	139675	114955
8 waiters	1E+06	511207	335178	247347	189047	153060	132393	118271

!atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	62262	61143	63176	62353	62338	62377	63545	63185
2 waiters	76468	80924	77032	82316	82086	82033	84138	75966
3 waiters	83437	83374	82644	81830	83382	82322	83600	83705
4 waiters	66871	87444	87234	89427	85681	88254	87496	88889
5 waiters	76252	77940	75874	76025	76158	77174	76622	77022
6 waiters	71377	70866	72501	71940	72910	71759	72147	72493
7 waiters	70143	63809	63293	70793	70349	70682	68398	63410
8 waiters	69008	68578	64773	66854	67183	67828	63632	68405

ReFS

lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1778	885	575	433	337	289	233	193
2 waiters	1721	856	274	172	137	96	82	75
3 waiters	1343	817	230	128	95	64	53	39
4 waiters	1350	746	185	108	60	53	32	29
5 waiters	1251	702	167	77	47	36	25	15
6 waiters	1170	336	135	66	38	22	15	14
7 waiters	1118	420	116	50	39	16	19	14
8 waiters	1077	263	108	54	26	15	15	10

byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1E+06	520213	356183	266872	199431	174761	131250	132080
2 waiters	147105	95502	43644	38065	34203	25537	34731	22051
3 waiters	211325	48373	151711	111803	74199	78343	68602	72383
4 waiters	167487	53039	58536	36708	32224	22506	22435	60353
5 waiters	220012	170473	128873	76189	30585	67130	46694	46701
6 waiters	88740	52341	44554	43073	31770	47076	50262	19321
7 waiters	203785	161033	127959	119133	81924	61040	22158	4103
8 waiters	84743	57221	55076	33044	79235	40358	5568	1733

atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	73318	73721	73507	73331	73902	73350	74353	73480
2 waiters	69587	53363	53536	55971	57507	52513	41748	52933
3 waiters	55658	52130	55622	52074	54543	55827	52924	56233
4 waiters	39485	40164	40753	40934	40163	41035	40735	41052
5 waiters	37040	37418	37183	37406	37250	37177	37011	37435
6 waiters	36234	36668	36250	36179	36148	36657	36232	36628
7 waiters	36000	36259	36216	36182	36083	36361	35936	36351
8 waiters	35151	35947	35440	35723	35414	35833	35638	35755

!lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1761	899	592	418	343	282	247	217
2 waiters	2156	1003	707	514	419	339	297	258
3 waiters	2150	1103	692	534	424	338	293	265
4 waiters	2224	1112	638	523	430	357	230	266
5 waiters	2220	1119	735	523	435	341	305	267
6 waiters	2353	1078	736	531	439	360	302	271
7 waiters	2372	1068	748	541	444	358	310	275
8 waiters	2348	1128	754	560	421	370	314	259

!byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1E+06	520411	355379	266540	211000	148776	151066	131804
2 waiters	1E+06	522655	135038	143839	235155	154452	90226	153741
3 waiters	194610	77004	35354	38425	25111	50603	43203	15568
4 waiters	132311	61955	113183	64402	40900	47368	27307	16595
5 waiters	350358	197264	92625	95750	66667	38116	43134	36534
6 waiters	332401	195857	87038	64157	54242	58424	38474	37733
7 waiters	333198	200036	129953	95895	58470	48473	51701	33208
8 waiters	427728	193048	127801	94760	74075	60952	42841	36861

!atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	73684	73841	73739	73841	73737	74479	74239	73135
2 waiters	59973	73369	64406	62279	77340	61615	66668	51723
3 waiters	63173	68034	60841	58848	68673	66559	68932	68809
4 waiters	63048	63258	63254	63073	63123	63038	60094	62986
5 waiters	55039	55300	55353	55254	55513	55300	55268	54761
6 waiters	55907	55666	55759	54260	55286	55173	55352	55362
7 waiters	55676	56561	56260	56486	55704	56395	56323	56190
8 waiters	56381	56457	57552	57459	57183	57317	56776	56886

FAT32

lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	4126	2050	1337	1007	797	663	566	435
2 waiters	3702	1372	1118	590	443	359	296	251
3 waiters	3258	1559	619	370	256	202	168	126
4 waiters	2367	931	330	235	153	108	86	68
5 waiters	2756	708	255	151	101	66	46	35
6 waiters	2613	479	208	111	63	41	35	23
7 waiters	2476	411	150	78	46	34	24	17
8 waiters	2330	364	124	60	34	25	16	12

byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1E+06	649646	433432	303246	232784	221984	188825	165264
2 waiters	257238	67785	64820	111818	144412	115606	67078	23677
3 waiters	66433	53225	110765	154250	138002	119058	102922	109026
4 waiters	168795	65317	57760	43707	48906	39333	38234	31888
5 waiters	230486	184083	164186	145454	132486	114239	107321	78113
6 waiters	174589	103261	86641	46551	45428	43734	83868	13660
7 waiters	226353	173508	163148	147564	128971	91736	47306	6143
8 waiters	151525	58334	51814	52853	50787	61537	6977	3293

atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	87128	87425	87593	88028	87932	87852	87442	87693
2 waiters	113249	111361	105834	104316	114361	112666	80450	78341
3 waiters	79563	73156	79160	78737	79639	79630	73329	79760
4 waiters	64640	64628	65525	64833	65592	64684	65234	64766
5 waiters	63313	63308	63176	63654	63184	63320	63642	63646
6 waiters	61629	61672	61770	61603	61484	61505	61514	61493
7 waiters	60468	60683	60708	60742	60722	60646	60773	61090
8 waiters	60108	60196	59838	60380	60454	60507	60290	59380

!lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	3845	1982	1294	932	774	641	550	473
2 waiters	5219	2563	1693	1262	938	834	713	621
3 waiters	5826	2865	1880	1333	1114	928	794	672
4 waiters	6044	2978	1963	1456	1166	962	815	713
5 waiters	6298	3075	2027	1509	1191	992	840	735
6 waiters	6373	3141	2066	1526	1213	999	861	740
7 waiters	6467	3151	2085	1553	1232	1025	864	747
8 waiters	6432	3187	2105	1561	1240	1018	862	746

!byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1E+06	651134	445110	335650	235706	227129	190568	141714
2 waiters	2E+06	732533	506289	370064	284546	225384	158069	153741
3 waiters	2E+06	807585	485385	338737	276625	221715	182034	158496
4 waiters	2E+06	748291	445838	360652	230967	183784	157505	136053
5 waiters	2E+06	693743	425676	297692	228880	179782	153433	135206
6 waiters	2E+06	630979	414241	291869	215476	178964	153737	131342
7 waiters	2E+06	675036	414863	290319	223037	177217	151001	132833
8 waiters	2E+06	661316	406077	285391	215080	177074	153112	130403

!atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	87162	87245	87554	87810	87317	87427	87141	86341
2 waiters	89044	112755	113221	114158	112896	114243	115829	112943
3 waiters	101447	102124	116248	102042	102238	101334	101496	101015
4 waiters	34863	95024	95791	96045	95350	95761	95345	
5 waiters	82169	81923	81901	81901	81746	81572	81428	
6 waiters	80914	76546	79837	81304	80305	80401	79508	80044
7 waiters	80592	81554	81143	80834	80464	75865	79965	79507
8 waiters	79782	81153	81920	81259	79089	79079	80158	79733

exFAT

lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	3726	1630	1038	819	651	543	463	410
2 waiters	2912	1625	810	397	315	252	213	185
3 waiters	2558	1207	367	231	178	136	113	38
4 waiters	2190	737	194	142	105	78	60	67
5 waiters	1886	551	141	93	53	46	35	32
6 waiters	1851	452	103	65	39	33	20	16
7 waiters	1635	215	81	45	27	17	14	5
8 waiters	1575	327	62	34	17	14	11	5

byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1E+06	638981	437970	320686	255046	180394	187836	165060
2 waiters	303808	144943	83636	111243	136074	66560	33030	25273
3 waiters	235183	181299	157596	145889	116250	108336	31487	104735
4 waiters	87836	62657	67041	48958	50980	37125	37315	28000
5 waiters	219385	188574	127543	44743	134631	116242	91379	83503
6 waiters	86217	89597	108783	81226	52036	32584	34573	2477
7 waiters	227004	176342	159881	143632	126400	31463	38449	4583
8 waiters	81512	54344	50445	48631	51223	31016	4036	5

atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	85689	86584	86522	86634	86116	87054	86466	87020
2 waiters	89396	114035	112316	89724	61888	89214	89526	116452
3 waiters	78858	80587	80336	80665	80088	80326	79865	79538
4 waiters	65129	64532	60735	63635	64987	64534	64710	65255
5 waiters	63975	61311	63435	63464	63318	63731	62666	63681
6 waiters	61740	61406	61574	61226	61363	61838	61466	61177
7 waiters	60873	60813	60331	60413	60442	60854	60754	60112
8 waiters	53610	53936	60264	59603	60102	53943	60156	59881

!lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	3254	1604	1065	802	643	534	458	399
2 waiters	4510	2152	1446	1073	855	710	606	528
3 waiters	5293	2595	1688	1258	988	817	696	563
4 waiters	5536	2678	1773	1314	1037	847	716	623
5 waiters	5741	2749	1816	1342	1056	870	731	632
6 waiters	5832	2786	1792	1358	1076	885	744	641
7 waiters	5863	2849	1872	1385	1092	882	741	643

!byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1E+06	656683	437435	323985	251656	139013	167644	163037
2 waiters	2E+06	780303	486963	362633	287461	214017	138630	155703
3 waiters	2E+06	795368	495761	356739	256000	216853	185515	160223
4 waiters	2E+06	704313	441650	293441	231439	182145	153363	134874
5 waiters	2E+06	701754	425566	288508	213805	179301	154379	131910
6 waiters	2E+06	691817	417782	288313	222022	176361	150580	132323
7 waiters	2E+06	665967	407809	287003	212315	175631	151625	129728
8 waiters	2E+06	647714	401734	273177	215794	172387	146291	123244

!atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	86370	85703	87204	86697	87233	86852	87493	87161
2 waiters	103122	100838	112204	94047	92625	94208	100943	113740
3 waiters	100335	102446	101729	101960	101811	101260	102097	101015
4 waiters	96164	96415	96510	96234	95838	96418	96236	95848
5 waiters	81547	82173	81938	81367	81865	81579	81410	
6 waiters	81341	81183	81335	81170	79920	80117	80867	80712
7 waiters	80628	81339	80443	80298	80998	81137	80365	79971
8 waiters	80423	81375	81602	80768	81155	79876	80363	80751

Thank you

And let the questions begin!

Github: https://github.com/ned14/boost.afio

Ref docs: https://ned14.github.io/boost.afio/