Racing the File System

Workshop

Niall Douglas

Contents:

What is a race?

What is a race?

Two types often confused:

1. *Data Race*
   - Usually failure to enforce correct ordering and/or visibility of reads and writes
   - Diagnosis often automatable e.g. clang thread sanitiser, valgrind helgrind etc
2. *Race condition*
   - Non-determinism produces incorrectness
   - Skilled programmer needed to diagnose

What is a race: Classic example

Thread 1	Thread 2
x = read position 0	x = read position 0
x = x + 1	x = x + 1
position 0 = write x	position 0 = write x

Position 0 is only incremented by 1 not 2!

What is a filing system race?

1. Concurrent i/o

Thread 1

```
int x, y;
preadv(fd, x, 0);


preadv(fd, y, 4);
```

Thread 2

```
int b[2];


pwritev(fd, b, 0);
```

Thread 1 gets mismatched x and y

2. Concurrent path changes

<u>Thread 1</u> | <u>Thread 2</u>

```
path="/niall/store";
fd1=open(path+"/file1");
                              rename("/niall", "/niall.old");
                              rename("/other", "/niall");
fd2=open(path+"/file2");
```

Thread 1 gets mismatched file1 and file2

3. Deleting a directory tree

Standard depth-first algorithm:

1. Enumerate directory contents
2. For every directory, recurse to step 1, then delete directory
3. For every file, delete file

This is correct for POSIX, **but INCORRECT for Microsoft Windows**

Deleting a directory tree on Windows

1. Enumerate directory contents
2. For every non-empty directory, recurse to step 1
3. For every file, try to rename to random name in %TEMP and then delete
4. For every empty directory, rename to random name in %TEMP and then delete
5. Loop the above until directory tree deleted

Q: Why is this the correct algorithm on Windows?

Deleting a directory tree on Windows

1. Enumerate directory contents
2. For every non-empty directory, recurse to step 1
3. For every file, try to rename to random name in %TEMP and then ~~delete~~ mark for later deletion
4. For every empty directory, rename to random name in %TEMP and then ~~delete~~ mark for later deletion
5. Loop the above until directory tree deleted (may take as long as any item opened without `FILE_SHARE_DELETE` is open)

Why do filing system races matter?

- There are many, many more places where file system races will bite you unexpectedly
  - Most programmers assume the file system to be static and unchanging and that they are the only actor working with files
- In fact, the file system is a pit of concurrency races, security holes and unexpected program failure
  - Such as …

4. Security: Time Of Check To Time Of Use (TOCTTOU)

```
        Thread 1                          Thread 2
if access(path) is ok {
                                link(otherpath, path);
  fd = open(path);
  write(fd, …);
```

Secure file written bypassing security!

TOCTTOU even gets its own CWE ...

https://cwe.mitre.org/data/definitions/367.html

- CVE-2003-0813 RPC Denial of Service attack
- CVE-2004-0594 PHP arbitrary code execution
- CVE-2008-2958 Arbitrary file modify
- CVE-2008-1570 Arbitrary file modify

Portable race-free idioms and design patterns

Some basic design principles:

1. Avoid absolute paths like the plague, they are ALWAYS racy
2. Use an open file descriptor/HANDLE as the base for relative path operations using the special relative path system APIs
3. Combine the relative path system APIs with the design patterns presented next to achieve various race free behaviours

Special relative path file system APIs

POSIX (Linux, FreeBSD, OS X, many more) provides relative path replacements for absolute path taking system APIs:

- `execveat()`, `faccessat()`, `fchmodat()`, `fchownat()`, `fstatat()`, `futimesat()`, `linkat()`, `mkdirat()`, `mknodat()`, `openat()`, `readlinkat()`, `renameat()`, `symlinkat()`, `unlinkat()`, `utimensat()`

Special race free file system APIs

Microsoft Windows is a bit more tricky:

- All the NT kernel APIs can work from a base HANDLE to a directory to do relative path lookups like the POSIX `*at()` functions, sadly not exposed in Win32. However:
  - You cannot rename a directory containing any open file handle
  - You cannot rename files into any destination path where any process has an open HANDLE able to rename any part of that path
    - <u>**This prevents paths being changed during use**</u>

Race free design pattern 1:
Use Relative Paths instead of Absolute Paths

Design pattern 1: Relative paths

- Instead of:
  - fd1 = open("/niall/foo/file1");
  - fd2 = open("/niall/foo/file2");
- Do:
  - dirh = open("/niall/foo");
  - fd1 = openat(dirh, "file1");
  - fd2 = openat(dirh, "file2");

It is no longer possible to race file1 and file2 open!

Race free design pattern 2:

Avoid paths altogether by using direct-by-fd operations

Design pattern 2: Avoid paths altogether

- Instead of:
  - link("/niall/foo/file1", "/file2");
- Do:
  - linkat(file1_fd, "", AT_FDCWD, "/file2", AT_EMPTY_PATH);
- Some platforms (Linux, Windows) allow you to work directly from an open fd
  - This makes the operation completely race free

Design pattern 2: Avoid paths altogether

- What about instead of:
  - unlink("/niall/foo/file1");
- Do:
  - unlinkat(file1_fd, "", AT_EMPTY_PATH);
- Unfortunately this does not work on POSIX
  - Currently only Microsoft Windows allows delete-by-handle, so how do we work around this lack of support on POSIX?

Workaround design pattern 3:

Work around lack of direct-by-fd host OS support by combining relative path syscalls with inode checking

Design pattern 3: Inode checking

How do you workaround incomplete *filesystem-operation-from-fd* support?

- Windows, Linux, OS X and FreeBSD (directories only) permit you to ask for the current path of an open fd
- POSIX guarantees that a file with the same **`st_dev`** and **`st_ino`** values is the same file
- You can therefore reliably get **the parent directory** of some open fd on Linux, OS X and FreeBSD (directories only) using the **Inode checking** design pattern

Design pattern 3: Inode checking

Let’s say I have an open fd to any file located anywhere unpredictable. Steps:

1. Get current path of fd
2. Split path and open parent directory
3. Do fstatat(dirfd, leafname)
4. Compare `st_dev` and `st_ino`. If not ours, loop to step 1

Design pattern 3: Inode checking

- Once you reliably have an open fd to the parent directory, **<u>sibling lookups</u>** are trivial
  - i.e. for some open fd X to a file “store1.dat”, race free open fd to sibling file “store1.idx”
- You can also do race free deletion/rename
  - i.e. for some open fd X, race free delete/rename it on platforms not supporting direct-by-fd race free deletion/rename

Race free design pattern 4:

Use atomic renaming to prevent concurrent reads of partially completed writes

Design pattern 4: Atomic renaming

- A well known Unix design pattern for avoiding reader-writer visibility races is using atomic renaming:
  a. Create temp file with random/`O_TMPFILE` name and write data into it
  b. When complete, *atomically rename* the temp file to its canonical name, replacing the previous
  c. Users of the previous canonical version still see the previous inode which is deallocated on last fd close

Design pattern 4: Atomic renaming

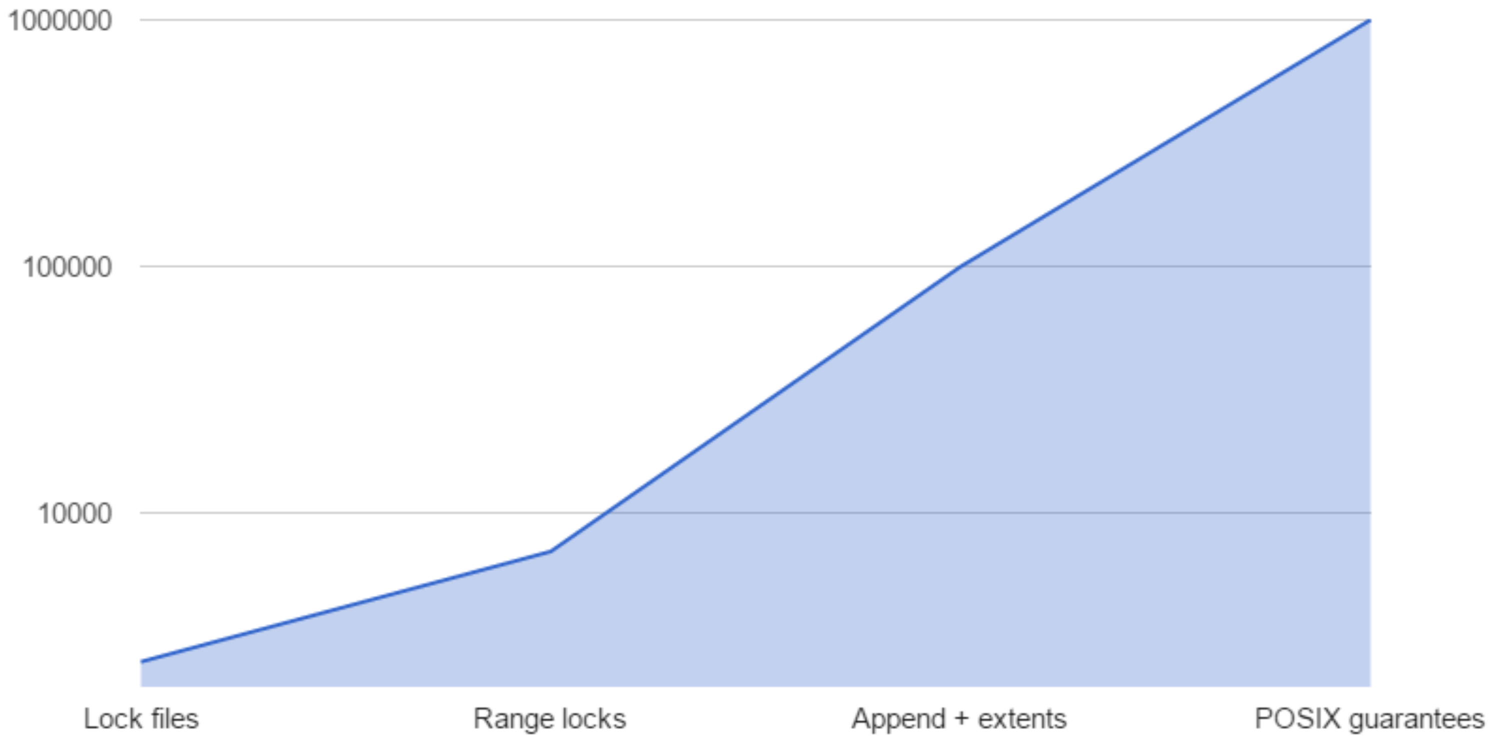
- Atomic renaming is traditionally avoided by portable code because Win32 did not provide it
  - The NT kernel **<u>always</u>** provided atomic renaming
  - And from Vista onwards, now so does Win32
- The name of the Win32 atomic rename API (NOT `MoveFileEx`!) is as unobvious as it gets:
  - SetFileInformationByHandle() with FILE_RENAME_INFO with ReplaceIfExists true

Race free design pattern 5:
The four techniques of concurrency control in the file system

Design pattern 5: Locking

There are four types of locking 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 Linux, tricky on non-Linux POSIX)
3. Atomic append + extent deallocation (extent based i. e. recent filing systems only)
4. Ordering guarantees (file system geeks only, probably only reliable on NTFS, XFS, ZFS, UFS)


1000000
100000
10000
Lock files
Range locks
Append + extents
POSIX guarantees

Design pattern 5a:
Exclusive lock files

Design pattern 5a: Lock files

- Exclusive lock files are easy:
  - while(-1==open("lockfile", O_EXCL|O_CREAT));
  - while(-1==CreateFile("lockfile", CREATE_NEW, FILE_ATTRIBUTE_TEMPORARY));
- Pros:
  - Works as expected on networked filing systems
  - Works as expected between operating systems on the same networked drive
  - Conceptually simple, so easy to maintain

Design pattern 5a: Lock files

Cons:

- Exclusive only i.e. cannot permit multiple readers
- Not sudden power loss friendly
- On POSIX, breaking stale lock files from unexpected process exit vs swap file thrashing is a problem
  - Windows has very useful delete-on-close facility
- No way of efficiently sleeping until a lock file is freed
  - Expensive on CPU and battery
- Performance is not great - 2.5k Windows O(log WAITERS), 4k Linux O(1), 10k FreeBSD O(1)

Design pattern 5b:
Byte range locks

Design pattern 5b: Byte range locks

- Byte range locks let you place an exclusive or a shared lock on some offset and length in an open file
- 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)
  - Much faster than lock files - 3.5k Linux O(waiters), 7k Windows O(1), 20k FreeBSD O(waiters)

Design pattern 5b: Byte range locks

Cons:

- Straightforward (and async!) to use on Windows, but painful to use on POSIX except Linux >= 3.15
  - On POSIX range locks are per inode, not per fd
  - On POSIX any single close() unlocks all locks for that inode for all fds in the process ☹
- Cross-platform byte range locks are problematic on shared networked drives
  - Advisory on POSIX, mandatory on Windows, plus on POSIX offset and length is signed unfortunately

Design pattern 5c:
Atomic append
+ Extent deallocation

Design pattern 5c: atomic append + extent deallocation

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

Design pattern 5c: 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
- Concurrency potentially > 100k IOPS all non-COW FSs but write complexity is O (waiters^X) where X is likely >= 2

Design pattern 5c: atomic append + extent deallocation

Pros:

- Very fast EXCEPT when concurrent appending + reading many small changes on copy-on-write filing systems
- Works well on all platforms, including multi-platform use of a CIFS network share
- 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)

Design pattern 5c: atomic append + extent deallocation

Cons:

- Requires an extent-based filing system (anything created in the past 15 years is usually extents-based) otherwise file grows forever
  - One can use *segmented files* to work around this
- Performs best if appended records are “chunky”
  - Extent granularity is anywhere between 4Kb and 128Kb depending on filing system
- Algorithms employed befuddles most (even otherwise excellent) engineers so maintenance can be a problem

Design pattern 5d:

POSIX concurrent change visibility ordering guarantees
(beware the dragons which abound here!)

Design pattern 5d: Ordering guarantees

For the true power programmer only …

- POSIX.2008 does provide some reader-writer change visibility ordering guarantees
- IF you are never on a networked drive AND (you are on BSD OR you are using XFS on Linux OR you are on Windows) …
    - … then this MAY work for you

Design pattern 5d: Ordering guarantees

POSIX.2008 says this:

> "I/O is intended to be atomic to ordinary files … Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations."
> (POSIX-2008)

This is *identical* to `std::memory_order_relaxed` for some `std::atomic<T>` where T is a some single preadv() or pwritev() operation!

Design pattern 5d: Ordering guarantees

POSIX.2008 also says this:

> *"If a read() of file data can be proven (by any means) to occur after a write() of the data, it must reflect that write(), even if the calls are made by different processes. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from write() calls to subsequent read() calls."* (POSIX-2008)

Design pattern 5d: Ordering guarantees

What does this mean?

- Every read() or readv() or preadv() for some offset and length implicitly excludes any concurrent write() or writev() or pwritev() overlapping the same offset and length
  - i.e. a write is NEVER seen partially completed by any read
  - This does NOT apply to all-reads nor all-writes!

Design pattern 5d: Ordering guarantees

- This *happens-before* *ordering guarantee* is similar to `std::memory_order_release` for pwritev() before `std::memory_order_acquire` for preadv() [with respect to write-before-read only]

i.e. **it can be used for lock-free algorithm programming same as std::atomic!**

Design pattern 5d: Ordering guarantees

Pros:

- About as fast as you can get > 1M IOPS
  - No locking at all beyond what the kernel does internally
- Conceptually familiar to anyone versed in lock-free atomics programming
- Works very well on any major operating system
  - … except Linux

Design pattern 5d: Ordering guarantees

Cons:

- Linux locks per 4Kb page only
  - You are sunk if your read or write straddles a 4Kb boundary - a work around is **`2^N`** record sizes
  - XFS on Linux adds extra locking so that does work
- This is not a well tested use case
  - No major database relies on this technique
  - Other POSIX (FreeBSD, Solaris) guarantees them
  - NT kernel + NTFS implements these semantics, but Microsoft make no guarantees this will remain

Introducing proposed Boost.AFIO

Proposed Boost.AFIO - what is it?

- Provides a single universal file system programming model
  - 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 - provides

- Race free filesystem API extending the Filesystem TS
- Abstracted reference counted open fd/handle model
- Potential arbitrary file system backends
  - `file:///` (your local hard drive)
  - `file:///foo.zip` (a ZIP archive)
  - `http://something/index.html` (HTTP)
- 98% asynchronous file system API
- 100% asynchronous scatter gather file i/o API
- Synchronous API equivalents in throwing and `error_code` variants

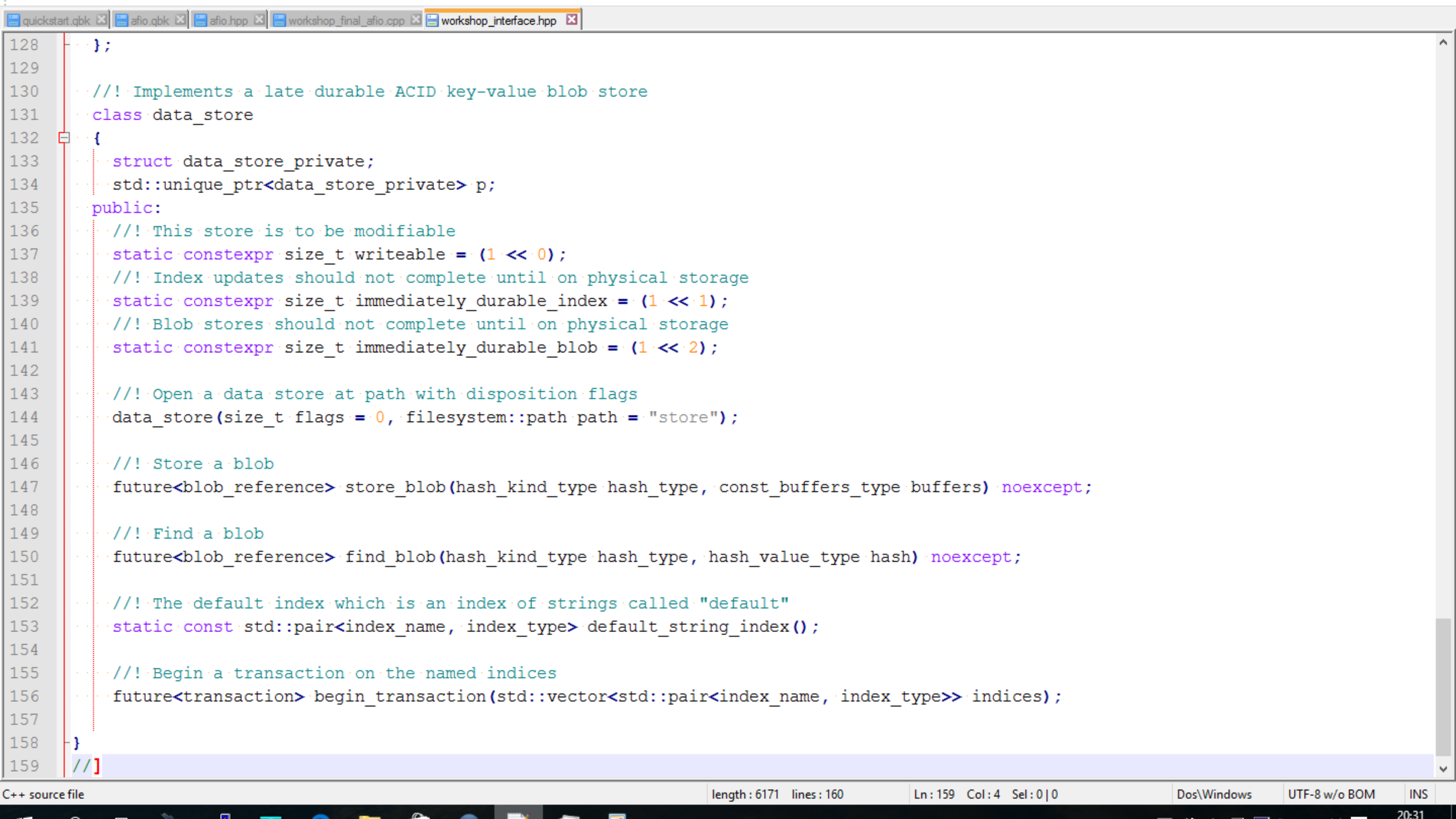
Proposed Boost.AFIO - APIs

- Open/create/delete file/directory/symlink relative to open fd/handle
- Sync to physical storage (3 algorithms)
- Deallocate physical storage via open fd/handle
- Atomic scatter read and gather write
- Examine mounted storage volume of open fd/handle
- Get current path of open fd/handle
- Get target of open fd/handle to symlink
- Map extents into memory
- Link/unlink open fd/handle relative to other open fd/handle
- Atomic rename of open fd/handle relative to other open fd/handle

Proposed Boost.AFIO - current status

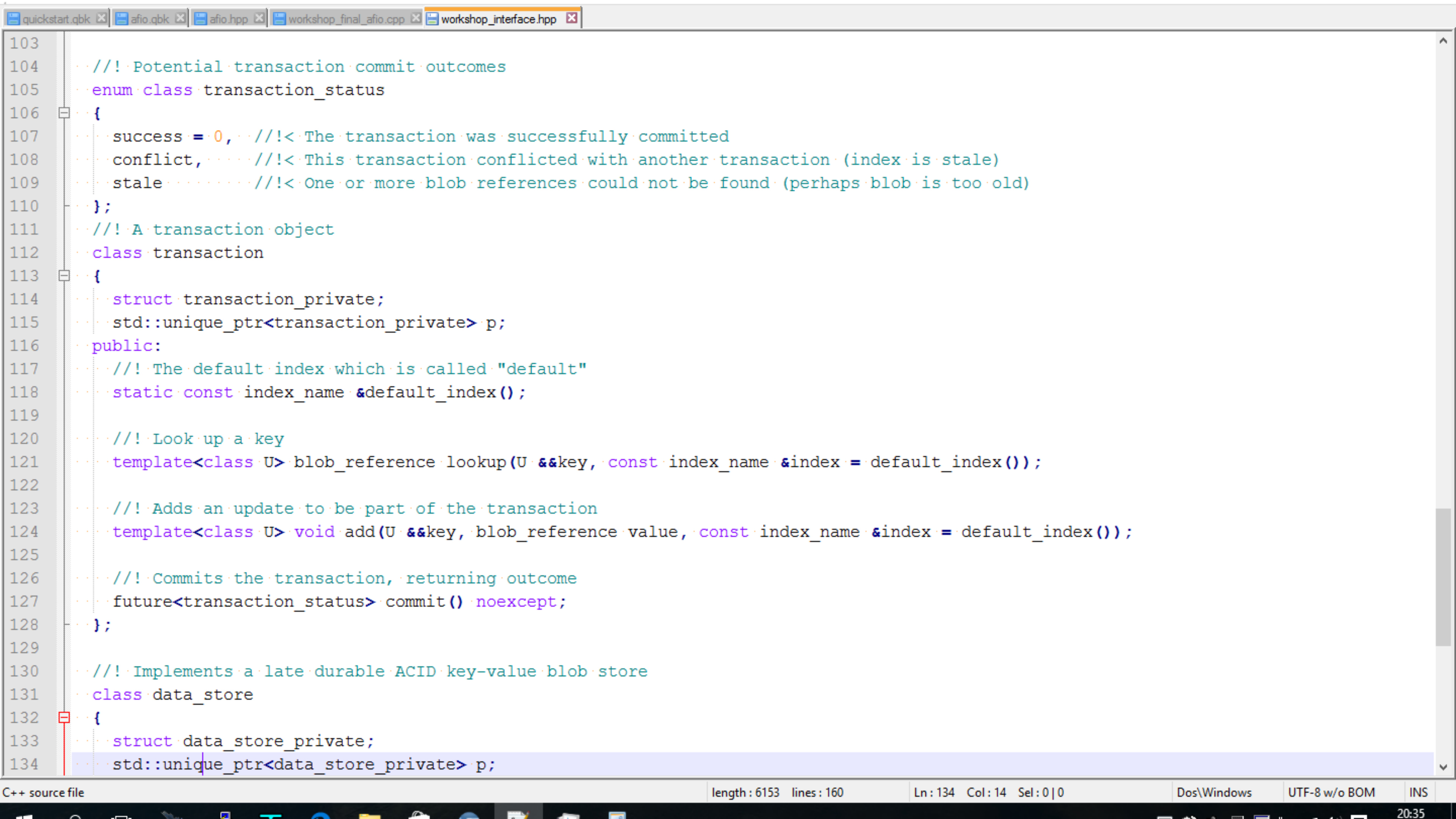
- Ported to Boost in 2013 by student Paul Kirth as part of Google Summer of Code
  - Entered Boost peer review queue in October 2013
- Was peer reviewed by Boost community August 2015
  - Universal rejection by all reviewers bar one
- Eventually will be rewritten using lightweight monadic futures + coroutines + post-ASIO i/o reactor
  - But existing <= C++0x-era engine is mature and end user API is not expected to change by much
  - New engine will just be lighter weight & C++ 1z ready
  - New non-ASIO i/o reactor makes feasible complete locking support

What can you do with this stuff?


quickstart.qbk | afio.qbk | afio.hpp | workshop_final_afio.cpp | workshop_interface.hpp

```cpp
  };

  //! Implements a late durable ACID key-value blob store
  class data_store
  {
    struct data_store_private;
    std::unique_ptr<data_store_private> p;
  public:
    //! This store is to be modifiable
    static constexpr size_t writeable = (1 << 0);
    //! Index updates should not complete until on physical storage
    static constexpr size_t immediately_durable_index = (1 << 1);
    //! Blob stores should not complete until on physical storage
    static constexpr size_t immediately_durable_blob = (1 << 2);

    //! Open a data store at path with disposition flags
    data_store(size_t flags = 0, filesystem::path path = "store");

    //! Store a blob
    future<blob_reference> store_blob(hash_kind_type hash_type, const_buffers_type buffers) noexcept;

    //! Find a blob
    future<blob_reference> find_blob(hash_kind_type hash_type, hash_value_type hash) noexcept;

    //! The default index which is an index of strings called "default"
    static const std::pair<index_name, index_type> default_string_index();

    //! Begin a transaction on the named indices
    future<transaction> begin_transaction(std::vector<std::pair<index_name, index_type>> indices);

}
//]
```

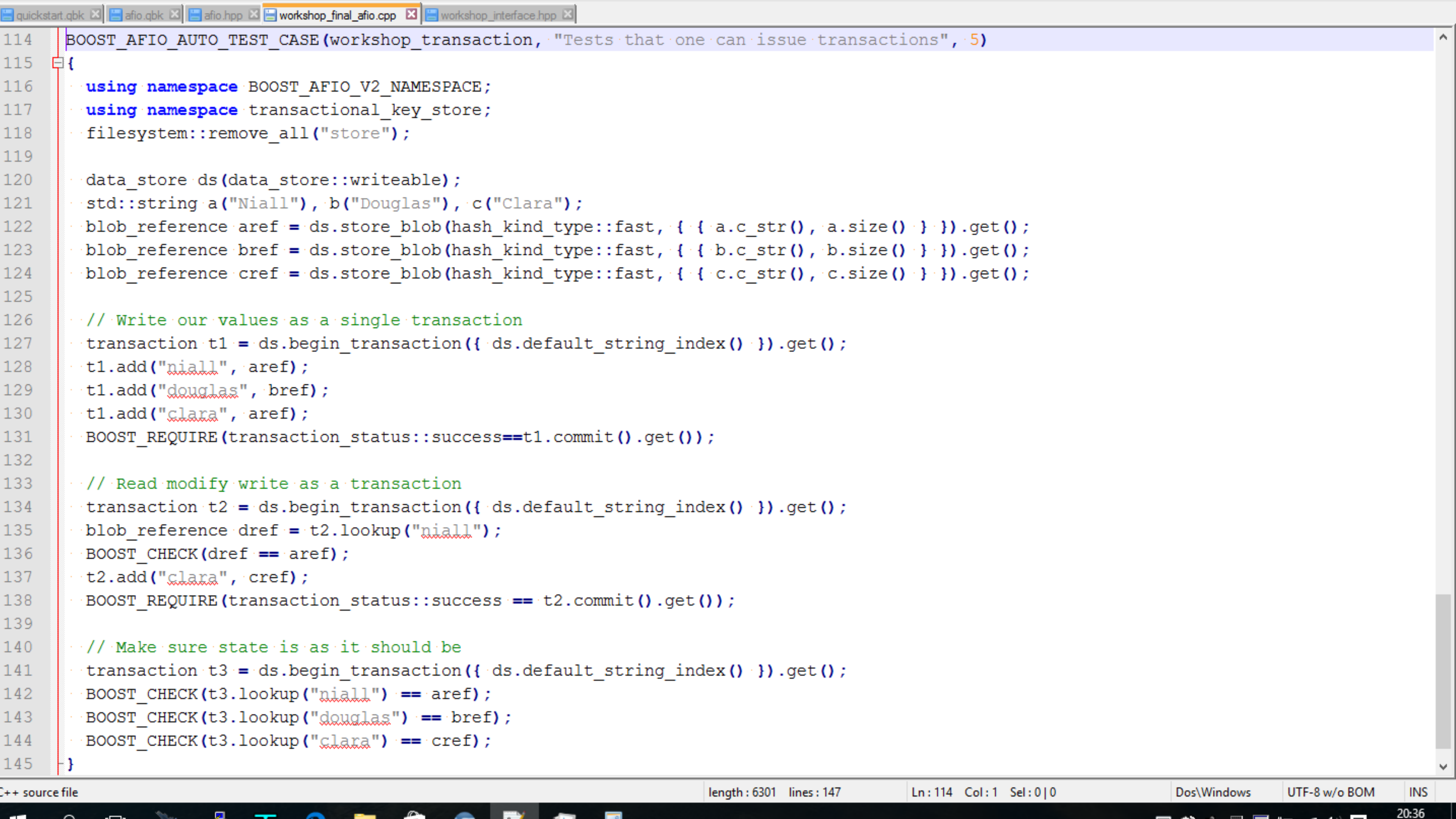
C++ source file   length : 6171   lines : 160   Ln : 159   Col : 4   Sel : 0 | 0   Dos\Windows   UTF-8 w/o BOM   INS



```cpp

  //! Potential transaction commit outcomes
  enum class transaction_status
  {
    success = 0,   //!< The transaction was successfully committed
    conflict,      //!< This transaction conflicted with another transaction (index is stale)
    stale          //!< One or more blob references could not be found (perhaps blob is too old)
  };
  //! A transaction object
  class transaction
  {
    struct transaction_private;
    std::unique_ptr<transaction_private> p;
  public:
    //! The default index which is called "default"
    static const index_name &default_index();

    //! Look up a key
    template<class U> blob_reference lookup(U &&key, const index_name &index = default_index());

    //! Adds an update to be part of the transaction
    template<class U> void add(U &&key, blob_reference value, const index_name &index = default_index());

    //! Commits the transaction, returning outcome
    future<transaction_status> commit() noexcept;
  };

  //! Implements a late durable ACID key-value blob store
  class data_store
  {
    struct data_store_private;
    std::unique_ptr<data_store_private> p;
```

C++ source file   length : 6153   lines : 160   Ln : 134   Col : 14   Sel : 0 | 0   Dos\Windows   UTF-8 w/o BOM   INS

20:35



```cpp
BOOST_AFIO_AUTO_TEST_CASE(workshop_transaction, "Tests that one can issue transactions", 5)
{
  using namespace BOOST_AFIO_V2_NAMESPACE;
  using namespace transactional_key_store;
  filesystem::remove_all("store");

  data_store ds(data_store::writeable);
  std::string a("Niall"), b("Douglas"), c("Clara");
  blob_reference aref = ds.store_blob(hash_kind_type::fast, { { a.c_str(), a.size() } }).get();
  blob_reference bref = ds.store_blob(hash_kind_type::fast, { { b.c_str(), b.size() } }).get();
  blob_reference cref = ds.store_blob(hash_kind_type::fast, { { c.c_str(), c.size() } }).get();

  // Write our values as a single transaction
  transaction t1 = ds.begin_transaction({ ds.default_string_index() }).get();
  t1.add("niall", aref);
  t1.add("douglas", bref);
  t1.add("clara", aref);
  BOOST_REQUIRE(transaction_status::success==t1.commit().get());

  // Read modify write as a transaction
  transaction t2 = ds.begin_transaction({ ds.default_string_index() }).get();
  blob_reference dref = t2.lookup("niall");
  BOOST_CHECK(dref == aref);
  t2.add("clara", cref);
  BOOST_REQUIRE(transaction_status::success == t2.commit().get());

  // Make sure state is as it should be
  transaction t3 = ds.begin_transaction({ ds.default_string_index() }).get();
  BOOST_CHECK(t3.lookup("niall") == aref);
  BOOST_CHECK(t3.lookup("douglas") == bref);
  BOOST_CHECK(t3.lookup("clara") == cref);
}
```

Thank you

And let the questions begin!