Arvados - Story #3198

[FUSE] Writable streaming arv-mount

07/07/2014 09:54 AM - Peter Amstutz

Status:	Resolved	**Start date:**	04/14/2015
Priority:	Normal	**Due date:**	
Assigned To:	Peter Amstutz	**% Done:**	100%
Category:	Keep	**Estimated time:**	0.00 hour
Target version:	2015-07-08 sprint		

Description

Writing output to Keep at the end of a job can be time consuming, especially if the network is slow, the Keep nodes are busy, or there is a very large amount of data. A writable arv-mount that could stream blocks as they are written could substantially improve performance.

Subtasks:

Task # 5718: Inode caching does not use unlimited memory	**Resolved**
Task # 6066: Review 3198-writable-fuse	**Resolved**
Task # 5445: Update FUSE to use new SDK	**Resolved**
Task # 5630: Write tests	**Resolved**
Task # 5719: Process event bus updates	**Resolved**
Task # 5895: Ensure vwd.checkin doesn't accidentally commit (or otherwise mess up) when...	**Resolved**
Task # 5785: Documentation and code cleanup	**Resolved**
Task # 5725: Review 3198-inode-cache	**Resolved**

Related issues:

Related to Arvados - Feature #4823: [SDKs] Good Collection API for Python SDK	**Resolved**	**12/17/2014**	
Related to Arvados - Bug #6194: [SDK] Support writing more than 64MiB at a ti...	**Resolved**	**06/01/2015**	
Related to Arvados - Bug #3833: [Keep] Improve file and directory invalidatio...	**Resolved**		
Has duplicate Arvados - Feature #4775: [SDKs] Add files to an existing Collec...	**Resolved**	**12/10/2014**	
Blocked by Arvados - Story #4930: [FUSE] Design: specify behavior for writabl...	**Resolved**	**01/28/2015**	
Precedes (1 day) Arvados - Story #6171: [FUSE] Document the writable FUSE mount	**Resolved**	**04/16/2015**	**04/16/2015**

Associated revisions

Revision c5909318 - 05/11/2015 03:57 PM - Peter Amstutz

Merge branch '3198-inode-cache' refs #3198

Revision 5b318755 - 06/25/2015 05:52 PM - Peter Amstutz

Merge branch '3198-writable-fuse' closes #3198

Revision 1ec9e370 - 06/25/2015 05:55 PM - Peter Amstutz

Update arvados_fuse version dependency on arvados-python-client. refs #3198

Revision a0c5e16c - 06/26/2015 09:09 PM - Ward Vandewege

Make sure to also build a backported package for python-llfuse. We need
it for writeable fuse.

refs #3198

Revision a0c5e16c - 06/26/2015 09:09 PM - Ward Vandewege

Make sure to also build a backported package for python-llfuse. We need
it for writeable fuse.

refs #3198

Revision 4f77c778 - 07/08/2015 09:19 PM - Brett Smith

6358: Declare FUSE driver's dependency on llfuse >= 0.40.

The workaround added in 08284382 requires this version. Below that,
llfuse's Queue at least lives in a different place. We may be able to
support older versions with more nuance, but for now, just codify the
current reality.

Closes #6358. Refs #3198.

History

#1 - 07/07/2014 09:56 AM - Peter Amstutz

- Description updated

#2 - 07/09/2014 04:52 PM - Tom Clegg

- Subject changed from Writable streaming arv-mount to [Keep] Writable streaming arv-mount

- Category set to Keep

#3 - 07/23/2014 04:23 PM - Peter Amstutz

Design proposal here:

https://arvados.org/projects/arvados/wiki/Writable_FUSE_mount

#4 - 08/08/2014 02:40 PM - Tom Clegg

- Target version set to Arvados Future Sprints

#5 - 09/19/2014 06:16 PM - Peter Amstutz

- Story points set to 5.0

#6 - 12/12/2014 01:52 PM - Peter Amstutz

- Target version changed from Arvados Future Sprints to 2015-01-07 sprint

#7 - 12/12/2014 02:03 PM - Peter Amstutz

- Assigned To set to Peter Amstutz

#8 - 12/12/2014 02:58 PM - Peter Amstutz

1. To update individual files
    1. On open, extract it to its own stream using as_manifest(). Create WritableFile inode entry.
    2. Set an "updated" flag in case the file is opened for writing but not actually updated.
    3. Create a buffer block and add it to the stream with a unique locator (not a content hash)
    4. Append each write to the buffer block, and update the file's locators and ranges to incorporate the write
    5. If a write will cause the buffer block to go over MAX_BLOCKSIZE, start a background thread to commit the block to Keep, create a new buffer block and add the write to the new block.
2. Directory modifications
    1. "rm" removes file from manifest
    2. "mv" extracts using as_manifest(), removes old and new names from original manifest, adds file back at the new name, normalizes
    3. "mkdir" creates new stream with zero length block and .keepdir file (not visible through FUSE)
    4. "rmdir" deletes a stream if it has no files (except .keepdir)
    5. "ln" works just like "mv" except that the original file is not removed. This is works nicely if the file is just being read, but if opened for writing this will have copy-on-write semantics, rather than modifying the same underlying inode. Consider poking POSIX in the eye since COW is arguably a more useful behavior.
3. On a full commit:
    1. Remove pending update files from the original manifest
    2. Combine pending update files into a single stream. Use a "." stream with full paths for each file
    3. Pack buffer blocks into a minimum number of Keep blocks, commit those blocks to Keep, and update stream locators and ranges
    4. Merge original manifest (minus updated files) with updated files stream and normalize
    5. Update collection record with new manifest
4. Automatically merging changes
    1. Add transactional update-if to API server (only commit if the base version is what we expected it to be)
    2. Take all new files
    3. Take non-conflicting updates.
    4. Take non-conflicting deletes. If deleted file is also modified, ignore delete.
    5. Write-write conflicts: preserve both versions. Newer update retains file name, older update is renamed.

#9 - 01/05/2015 03:22 PM - Peter Amstutz

- Target version changed from 2015-01-07 sprint to Arvados Future Sprints

#10 - 01/05/2015 10:22 PM - Tom Clegg

- Subject changed from [Keep] Writable streaming arv-mount to [DRAFT] [Keep] Writable streaming arv-mount

- Story points changed from 5.0 to 2.0

#11 - 01/16/2015 08:07 PM - Tom Clegg

- Target version changed from Arvados Future Sprints to 2015-02-18 sprint

#12 - 01/28/2015 07:25 PM - Peter Amstutz

Detailed semantics and discussion in #4930

#13 - 01/28/2015 07:28 PM - Peter Amstutz

- Story points changed from 2.0 to 3.0

#14 - 01/29/2015 06:22 PM - Tom Clegg

- Target version changed from 2015-02-18 sprint to Arvados Future Sprints

#15 - 02/03/2015 08:03 PM - Tom Clegg

- Subject changed from [DRAFT] [Keep] Writable streaming arv-mount to [FUSE] Writable streaming arv-mount

#16 - 02/06/2015 07:04 PM - Tom Clegg

- Target version changed from Arvados Future Sprints to 2015-03-11 sprint

#17 - 02/18/2015 08:22 PM - Ward Vandewege

- Target version changed from 2015-03-11 sprint to Arvados Future Sprints

#18 - 02/18/2015 08:43 PM - Ward Vandewege

- Target version changed from Arvados Future Sprints to 2015-04-01 sprint

#19 - 03/27/2015 05:15 PM - Tom Clegg

- Target version changed from 2015-04-01 sprint to 2015-04-29 sprint

#20 - 03/27/2015 06:27 PM - Tom Clegg

- Assigned To deleted (Peter Amstutz)

#21 - 04/01/2015 07:24 PM - Peter Amstutz

- Assigned To set to Peter Amstutz

#22 - 04/14/2015 07:23 PM - Peter Amstutz

3198-inode-cache lays groundwork for writable FUSE.

- Refactors classes from one big file into fresh.py, fusefile.py and fusedir.py
- Adds a framework for maintaining an LRU cache of directories. When a directory expires its contents are cleared out (but may be reloaded on demand later). This is only a soft memory cap but makes in my empirical tests a huge difference in mitigating the problem of unchecked growth in memory usage.

#23 - 04/22/2015 02:56 PM - Peter Amstutz

Additional notes on 3198-inode-cache:

File and all of its subclasses moved to fusefile.py. Mostly unmodified except for the addition of inc_use(), dec_use() and no-op clear() methods.

Directory and all of its subclasses moved to fusedir.py. Added inc_use() and dec_use() methods. Modified clear() method of CollectionDirectory to clean up recursively. Other classes are unchanged. Added persisted() flag to indicate whether a directory supports being cleared and reloaded later (right now only CollectionDirectory is persisted(), but other directory types could be as well.)

In *init*.py, added InodeCache. Inodes which are persisted() are added to the inode cache. When the cache exceeds desired capacity, start clearing the contents of inodes in cache until it is under the limit.

The use count indicates when an inode is currently in use (for example, there is an open directory handle), this prevents it from being cleared from the cache until the use count drops to zero.

Before you mention it, I'm already planning on changing it so that "_cache_priority" and "_cache_size" size are public fields of "FreshBase" and

InodeCache is not accessing private fields

#24 - 04/22/2015 09:56 PM - Brett Smith

Reviewing 3198-inode-cache at 2139ee84

- Any chance of getting tests for this? I know it's difficult to integration test since it only changes performance characteristics, but InodeCache seems isolated well enough that you could unit test it. Just pass in mock objects to manage() and unmanage() and make sure they get cleared or not appropriately.
- The term "inode cache" is perfectly accurate from a code perspective, but seems awfully technical to expose through the interface (i.e., the --inode-cache flag). Would another name maybe help users understand its purpose a little better? Just brainstorming a few ideas: listing cache, tree cache, entries cache. This is not that critical since we can always just rename it/add an alias later, but I at least wanted to flag it and see if there's another name you like as well.
- Where do the multipliers in objsize() methods come from? (128 in CollectionDirectory, 1024 in ProjectDirectory)
- Instead of passing in an inode cache size to Inodes.__init__(), why not just pass in an instantiated InodeCache directly? This would make it easier to grow InodeCache in the future without having to worry about keeping Inodes in sync with it. Similarly for Operations. (This provides a good illustration of what I mean: you're having to update Operations in this branch, just because you want to add constructor arguments to Inodes.)
- Why did RecursiveInvalidateDirectory lose its lock management?
- There's a noop need_gc = False in InodeCache.cap_cache().
- What's up with changes to the Python SDK to pass 0 to Range constructors where it's already a default argument? It doesn't seem to be a real problem, but it also seems to be a noop, which makes me wonder why it's here.

Thanks.

#25 - 04/24/2015 08:06 PM - Peter Amstutz

- Status changed from New to In Progress

- Story points changed from 3.0 to 5.0

#26 - 04/29/2015 07:08 PM - Peter Amstutz

- Target version changed from 2015-04-29 sprint to 2015-05-20 sprint

#27 - 04/29/2015 07:08 PM - Peter Amstutz

- Story points changed from 5.0 to 2.0

#28 - 05/07/2015 08:14 PM - Peter Amstutz

Brett Smith wrote:

> Reviewing 3198-inode-cache at 2139ee84
>
> - Any chance of getting tests for this? I know it's difficult to integration test since it only changes performance characteristics, but InodeCache seems isolated well enough that you could unit test it. Just pass in mock objects to manage() and unmanage() and make sure they get cleared or not appropriately.

Done.

> - The term "inode cache" is perfectly accurate from a code perspective, but seems awfully technical to expose through the interface (i.e., the --inode-cache flag). Would another name maybe help users understand its purpose a little better? Just brainstorming a few ideas: listing cache, tree cache, entries cache. This is not that critical since we can always just rename it/add an alias later, but I at least wanted to flag it and see if there's another name you like as well.

Now --file-cache (for the block cache) and --directory-cache (for the metadata cache).

> - Where do the multipliers in objsize() methods come from? (128 in CollectionDirectory, 1024 in ProjectDirectory)

Empirically derived heuristic (I fiddled with it until it seemed to be the right order of magnitude).

Removed objsize() from ProjectDirectory because it's not currently being used.

> - Instead of passing in an inode cache size to Inodes.__init__(), why not just pass in an instantiated InodeCache directly? This would make it easier to grow InodeCache in the future without having to worry about keeping Inodes in sync with it. Similarly for Operations. (This provides a good illustration of what I mean: you're having to update Operations in this branch, just because you want to add constructor arguments to Inodes.)

Done.

> - Why did RecursiveInvalidateDirectory lose its lock management?

It was obsolete code from a previous approach to updating directory contents based on event bus events (the rest of which was removed awhile ago). The main 3198 writable fuse branch will have a better approach.

> - There's a noop need_gc = False in InodeCache.cap_cache().

Done.

> - What's up with changes to the Python SDK to pass 0 to Range constructors where it's already a default argument? It doesn't seem to be a real problem, but it also seems to be a noop, which makes me wonder why it's here.

I was auditing memory usage and one of the things I tried was changing the Range object to be a named tuple to see if that was lighter weight than a regular object. Since named tuples can't have default values I had to add the extra value in. It didn't seem to make a difference in memory usage so I backed it out, but left in the extra parameter on the constructors in case we wanted to revisit.

#29 - 05/11/2015 02:45 PM - Brett Smith

0c01dc22 is good to merge. A few final thoughts.

Peter Amstutz wrote:

> Brett Smith wrote:
>
> > Reviewing 3198-inode-cache at 2139ee84
> >
> > - Any chance of getting tests for this? I know it's difficult to integration test since it only changes performance characteristics, but InodeCache seems isolated well enough that you could unit test it. Just pass in mock objects to manage() and unmanage() and make sure they get cleared or not appropriately.
>
> Done.

A few thoughts about these:

- The tests repeatedly set entN.return_value.in_use - should that be entN.in_use.return_value?
- There's a comment that says "ent3 is persisted, adding it should cause ent1 to get cleared," but then the next couple of assertions we do check that ent1.clear has *not* been called. I think the code is fine, but on a first pass this was a little confusing. Would it maybe help clarify things to move this comment down to around "Change min_entries" where ent1 does get cleared?
- If you have time, it would be great to break this down into smaller test cases. Being able to run multiple tests and see which ones are passing and which ones are failing can help diagnose where the problem is. With one omnibus test, it sometimes happens that you fix one issue only to reveal another test failure further down.

> > - Where do the multipliers in objsize() methods come from? (128 in CollectionDirectory, 1024 in ProjectDirectory)
>
> Empirically derived heuristic (I fiddled with it until it seemed to be the right order of magnitude).

Adding a small comment to explain how you arrived at the number would probably be nice for future readers.

> > - Instead of passing in an inode cache size to Inodes.__init__(), why not just pass in an instantiated InodeCache directly? This would make it easier to grow InodeCache in the future without having to worry about keeping Inodes in sync with it. Similarly for Operations. (This provides a good illustration of what I mean: you're having to update Operations in this branch, just because you want to add constructor arguments to Inodes.)
>
> Done.

This code instantiates an InodeCache directly in Operations' __init__ signature. Having a mutable argument in a function signature is almost always something to avoid in Python, since future callers will use the same mutated object. Having the default be None and instantiating the InodeCache in the function body would be more idiomatic. I realize it's a little academic in this case since it's hard to imagine why real code would instantiate more than one Operations, but I think this change would help reassure future readers that you're not doing something clever like intentionally sharing an InodeCache across instances.

Thanks.

#30 - 05/11/2015 03:17 PM - Peter Amstutz

Brett Smith wrote:

> A few thoughts about these:

- The tests repeatedly set entN.return_value.in_use - should that be entN.in_use.return_value?

That's a mistake, thanks for catching it.

- There's a comment that says "ent3 is persisted, adding it should cause ent1 to get cleared," but then the next couple of assertions we do check that ent1.clear has *not* been called. I think the code is fine, but on a first pass this was a little confusing. Would it maybe help clarify things to move this comment down to around "Change min_entries" where ent1 does get cleared?

Those assertions are unnecessary, so I removed them.

- If you have time, it would be great to break this down into smaller test cases. Being able to run multiple tests and see which ones are passing and which ones are failing can help diagnose where the problem is. With one omnibus test, it sometimes happens that you fix one issue only to reveal another test failure further down.

I broke things into a few smaller tests, although the result is more code overall.

> > - Where do the multipliers in objsize() methods come from? (128 in CollectionDirectory, 1024 in ProjectDirectory)
> >
> > Empirically derived heuristic (I fiddled with it until it seemed to be the right order of magnitude).
>
> Adding a small comment to explain how you arrived at the number would probably be nice for future readers.

Done.

> > - Instead of passing in an inode cache size to Inodes.__init__(), why not just pass in an instantiated InodeCache directly? This would make it easier to grow InodeCache in the future without having to worry about keeping Inodes in sync with it. Similarly for Operations. (This provides a good illustration of what I mean: you're having to update Operations in this branch, just because you want to add constructor arguments to Inodes.)
> >
> > Done.
>
> This code instantiates an InodeCache directly in Operations' __init__ signature. Having a mutable argument in a function signature is almost always something to avoid in Python, since future callers will use the same mutated object. Having the default be None and instantiating the InodeCache in the function body would be more idiomatic. I realize it's a little academic in this case since it's hard to imagine why real code would instantiate more than one Operations, but I think this change would help reassure future readers that you're not doing something clever like intentionally sharing an InodeCache across instances.

That is definitely not what I intended, thanks for the bit of Python language lawyering. Fixed.

#31 - 05/11/2015 03:37 PM - Brett Smith

The test failure I mentioned on IRC was a false alarm, caused by bad database state due to a badly-timed ^C. No worries there.

Peter Amstutz wrote:

> Brett Smith wrote:
>
> > - There's a comment that says "ent3 is persisted, adding it should cause ent1 to get cleared," but then the next couple of assertions we do check that ent1.clear has *not* been called. I think the code is fine, but on a first pass this was a little confusing. Would it maybe help clarify things to move this comment down to around "Change min_entries" where ent1 does get cleared?
>
> Those assertions are unnecessary, so I removed them.

I'm not sure we're talking about the same assertions? At 184dabe, it's still the case that the next couple of assertions on ent1 after this comment are self.assertFalse(ent1.clear.called).

I noticed there are a couple of places where you set entN.clear.called = False. I'm not sure how well this is supported by the Mock API, although obviously it's working fine now. Are you maybe looking for reset_mock? Or you could assert against the mock's call_count instead.

But both of these are little cleanup things that can either be merged directly, or done later. Thanks.

#32 - 05/18/2015 06:59 PM - Peter Amstutz

3198-writable-fuse is ready for review. Notes:

- Supports creating, writing, renaming, deleting files in collections
- Supports creating, renaming, deleting directories in collections
- Supports moving files and directories between collections
- Supports POSIX behavior allowing access to files which are unlinked but have open file handles
- Supports POSIX behavior allowing directories to move without invalidating open file handles
- Supports creating and deleting directories in projects (creates/deletes collection records)
- Supports moving collections (collection records) between project directories

Not supported

- Moving subdirectories of a collection into a project (would require creating a new collection record)
- Moving a collection out of a project into collection as a subcollection
- Editing or creating files representing records, such as pipeline templates

#33 - 05/19/2015 07:01 PM - Tom Clegg

Suggest changing "Collection must be writable" error message to "Collection is not writable" or "Collection is read-only". (Less confusing to simply state the condition that caused an error, e.g., ENOENT is "does not exist", not "must exist"...)

Ditto for

- "Path must refer to a file." → "Is a directory" for EISDIR.
- "Interior path components must be subcollection." → "Not a directory" (or "Not a directory: %s"?) for ENOTDIR.
- ...well, anyway, just search for error messages with "must" and replace with an appropriate "is not".

In commit_bufferblock(self, block, wait) perhaps wait should be called "sync" instead? ("Wait" seems a bit indirect, and wait=False waits sometimes.) Also, comment has "unless if", should be just "unless".

- When calling I think it's worth saying commit_bufferblock(self._current_bblock, sync=False) for booleans, instead of just commit_bufferblock(self._current_bblock, False). Clearer ("False what?"), and helps avoid callers falling out of sync with a changing function signature.
- Ditto for flush(True) and flush(False). And _reparent() would be a bit clearer if it said explicitly self.flush(sync=True)

The "start threads" part of commit_bufferblock looks like it should be moved out into its own @synchronized method like stop_threads.

commit_all's docstring should be updated now that commit_bufferblock can be synchronous too. ("like commit_bufferblock(sync=True), ..."?)

In commit_buffer_block, seemingly unprotected by self.lock or @synchronized:

```
if block.state() == _BufferBlock.WRITABLE:
    # ...
    block.set_state(_BufferBlock.PENDING)
```

Two questions:

1. Would if block.state() == block.WRITABLE be nicer? (Personally I like to avoid referring to block's exact type when I don't need to -- not sure if one way is more Pythonic)
2. This seems to create races and I'm not sure they'll never result in harmful/wasteful/confusing results. state() and set_state() themselves are @synchronized, but we seem to be assuming the state won't change between those two calls. Should we wrap that intervening code in self.lock?
    - commit_bufferblock() seems to be called only from other @synchronized methods -- is this meant to be part of commit_bufferblock's API? In that case, its docstring should warn about this, and it should be _privatized because callers other than self can't really use it safely.

Similarly, commit_all *doesn't* seem to have any internal or external locking, so it looks like self._bufferblocks could have WRITABLE items in it by the time we get to flush(True). Is this OK?

ArvadosFile.flush()'s docstring says it "flushes bufferblocks to Keep", but it looks like it is only capable of flushing the "current" bufferblock to keep. If wait, it also checks whether any other bufferblocks are uncommitted; if so, it prints a log message but still reports success (should this be an exception?). Is the docstring wrong? (My impression is that the put-queue can be non-empty at any given moment so we can't assume the current bufferblock is the only one that could be uncommitted.)

ArvadosFile.flush() will be more readable IMO if it handles "if not self.modified(): return" at the top and then outdents the rest of the code.

This seems like a strange restriction, and it shouldn't be necessary. Let's do the obvious right thing (like in read()) instead of failing:

```
        if len(data) > config.KEEP_BLOCK_SIZE:
            raise ArgumentError("Please append data in chunks smaller than %i bytes (config.KEEP_BLOCK_SIZE)"
 % (config.KEEP_BLOCK_SIZE))
```

The docstrings for ArvadosFile.set_unmodified and ArvadosFile.modified aren't very helpful. "Test the modified flag", sure, but what is the significance of the modified flag? AFAICT, file.modified() becomes True the first time a file is modified and stays True (regardless of close(), flush(), etc) until the parent collection is committed. Is this the intended meaning? It seems a bit more subtle than the intuitive definition of "modified". I wonder whether it would be more clear to call this flag "committed" (or "uncommitted") instead?

Likewise the docstring for RichCollectionBase.modified could be a bit more helpful (modified since when?).

The maxsize of the _put_queue should be a class constant. (And I suppose num_put_threads and num_get_threads might as well be class variables or even class constants, until/unless there's some motivation/mechanism for changing them at runtime.)

The "mode" argument to ArvadosFileReader seems superfluous: the only mode that makes sense is "r". Perhaps the argument should be omitted, or (since having the same signature seems nice) it might be worth throwing an exception if mode is not "r", instead of just ignoring it.

FileWriter and FileReader are documented to be thread-unsafe, but it seems like we could throw self.lock = threading.RLock() and a @synchronized decorator on the methods that use _filepos, and they'd become thread-safe. Is this correct? (This probably shouldn't hold up the branch if it's not as easy as that.)

This seems to be a pre-existing bug, but: RichCollectionBase.mkdirs is like os.makedirs() (not os.mkdirs()), but unlike os.makedirs() does not raise an exception if the target already exists. It also returns an ArvadosFile (instead of raising an exception as one would expect) if the target already exists and is a file.

(tbc)

#34 - 05/20/2015 07:09 PM - Brett Smith

- Target version changed from 2015-05-20 sprint to 2015-06-10 sprint

#35 - 05/21/2015 03:45 PM - Tom Clegg

I think conflict filenames should avoid using ":" characters. They have a tendency to confuse programs that know about host.name:dir/file syntax. How about just YYYY-MM-DD-HHMMSS? Perhaps we could also use something like "foo~timestamp~conflict~" instead of "foo~conflict-timestamp~" so users/scripts can safely remove "*~conflict~"? (Globs like "*~conflict*~" seem less safe, if only slightly...)

I find the test cases hard to follow because "file1", "file2", "collection1", "collection2", "foo", and "bar" don't tell me what role they play in the test scenario. For example, perhaps fuseWriteFileTestHelper1 and fuseWriteFileTestHelper2 could be called fuseWriteHelloWorldToFile1 and fuseReadHelloWorldFromFile1...?

Can we move the fuse*Helper functions inside the test classes they serve? I think that would make it much easier to see how the code fits together.

I don't think this test has any business asserting the exact format of a permission signature, the absence of additional hints, etc. I'd say if it matches ...a20(\+\S+)? 0:12:... then fuse performed correctly.

```
+        self.assertRegexpMatches(collection2["manifest_text"],
+            r'\. 86fb269d190d2c85f6e0468ceca42a20\+12\+A[a-f0-9]{40}@[a-f0-9]{8} 0:12:file1\.txt$')
```

Each test case takes about 4 seconds, most of which (I hope!) is setup/teardown. Can we do better? We could easily reduce the 0.5 second granularity in the unmount wait loop, but the setup seems even more expensive. Can we leave the mount alive across several test cases, and use some other way to give each test case a sufficiently blank state? (Although test independence is valuable, it would also be somewhat reassuring to test that the mount keeps working for longer than a single test case.) Perhaps we can also speed up the mounting process?

I'd like to see at least one stress test that could conceivably expose unanticipated race conditions. We could even get an interesting performance data point while we're at it. Perhaps something like "start N processes, each of which does M iterations of {copy file1.txt to a new collection called "test%d", commit, write different stuff, commit, try to read one of the other process's "test%d" collection}".

There's a lot of "Calling fuse_mount" spam in fuse tests. I'm guessing it's because of this. Is this left in intentionally?

```
llogger.setLevel(logging.DEBUG)
```

Please uncomment or remove:

```
-        _logger.debug("total is %i cap is %i", self._total, self.cap)
+        #_logger.debug("InodeCache total is %i cap is %i", self._total, self.cap)
```

I think this should be self.events:

```
        self.event = arvados.events.subscribe(api_client,
```

Is this logging.DEBUG line left in intentionally?

```
class FuseUpdateFromEventTest(MountTestBase):
    def runTest(self):
        arvados.logger.setLevel(logging.DEBUG)
```

Is there some situation where a non-collection object_uuid is in inode_cache? The "object_kind" condition here seems a bit surprising:

```
                item = self.inodes.inode_cache.find(ev["object_uuid"])
                if item is not None:
                    item.invalidate()
                    if ev["object_kind"] == "arvados#collection":
                        item.update(to_pdh=ev.get("properties", {}).get("new_attributes", {}).get("
portable_data_hash"
                    else:
                        item.update()
```

This is off-putting at best:

```
Ran 30 tests in 116.204s

OK
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "/usr/lib/python2.7/multiprocessing/util.py", line 284, in _exit_function
    info('process shutting down')
TypeError: 'NoneType' object is not callable
Error in sys.exitfunc:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "/usr/lib/python2.7/multiprocessing/util.py", line 284, in _exit_function
    info('process shutting down')
TypeError: 'NoneType' object is not callable
```

I tried fixing it by putting self.pool.join() before self.pool=None (as the multiprocessing docs seem to suggest) but that just adds a bunch of these messages before the atexit stuff:

```
Exception TypeError: TypeError("'NoneType' object does not support item deletion",) in <Finalize object, dead>
 ignored
```

Does this mean the processes really aren't exiting cleanly for some reason? The test-case-within-multiprocessing-within-test-case stuff seemed a bit sketchy; perhaps "move helpers into test classes" will actually help Python, not just devs?

(tbc)

#36 - 06/10/2015 07:08 PM - Brett Smith

- Target version changed from 2015-06-10 sprint to 2015-07-08 sprint

#37 - 06/11/2015 07:31 PM - Peter Amstutz

Tom Clegg wrote:

> Suggest changing "Collection must be writable" error message to "Collection is not writable" or "Collection is read-only". (Less confusing to simply state the condition that caused an error, e.g., ENOENT is "does not exist", not "must exist"...)

Done.

> In commit_bufferblock(self, block, wait) perhaps wait should be called "sync" instead? ("Wait" seems a bit indirect, and wait=False waits sometimes.) Also, comment has "unless if", should be just "unless".

Done.

> - When calling I think it's worth saying commit_bufferblock(self._current_bblock, sync=False) for booleans, instead of just commit_bufferblock(self._current_bblock, False). Clearer ("False what?"), and helps avoid callers falling out of sync with a changing function signature.
> - Ditto for flush(True) and flush(False). And _reparent() would be a bit clearer if it said explicitly self.flush(sync=True)

Done.

> The "start threads" part of commit_bufferblock looks like it should be moved out into its own @synchronized method like stop_threads.

Done.

> commit_all's docstring should be updated now that commit_bufferblock can be synchronous too. ("like commit_bufferblock(sync=True), ..."?)

Done.

> In commit_buffer_block, seemingly unprotected by self.lock or @synchronized:
> [...]
>
> Two questions:
>
> 1. Would if block.state() == block.WRITABLE be nicer? (Personally I like to avoid referring to block's exact type when I don't need to -- not sure if one way is more Pythonic)
> 2. This seems to create races and I'm not sure they'll never result in harmful/wasteful/confusing results. state() and set_state() themselves are @synchronized, but we seem to be assuming the state won't change between those two calls. Should we wrap that intervening code in self.lock?
>   - commit_bufferblock() seems to be called only from other @synchronized methods -- is this meant to be part of commit_bufferblock's API? In that case, its docstring should warn about this, and it should be _privatized because callers other than self can't really use it safely.
>
> Similarly, commit_all *doesn't* seem to have any internal or external locking, so it looks like self._bufferblocks could have WRITABLE items in it by the time we get to flush(True). Is this OK?
>
> ArvadosFile.flush()'s docstring says it "flushes bufferblocks to Keep", but it looks like it is only capable of flushing the "current" bufferblock to keep. If wait, it also checks whether any other bufferblocks are uncommitted; if so, it prints a log message but still reports success (should this be an exception?). Is the docstring wrong? (My impression is that the put-queue can be non-empty at any given moment so we can't assume the current bufferblock is the only one that could be uncommitted.)

Refactored bufferblock handling to be more correct:

- No more testing state/setting state
- Added an retryable "error" state, so if a block fails PUT it's possible to call commit_block() again

> ArvadosFile.flush() will be more readable IMO if it handles "if not self.modified(): return" at the top and then outdents the rest of the code.

Done.

> This seems like a strange restriction, and it shouldn't be necessary. Let's do the obvious right thing (like in read()) instead of failing:

Fixed in #6194

> The docstrings for ArvadosFile.set_unmodified and ArvadosFile.modified aren't very helpful. "Test the modified flag", sure, but what is the significance of the modified flag? AFAICT, file.modified() becomes True the first time a file is modified and stays True (regardless of close(), flush(), etc) until the parent collection is committed. Is this the intended meaning? It seems a bit more subtle than the intuitive definition of "modified". I wonder whether it would be more clear to call this flag "committed" (or "uncommitted") instead?

Renamed to "committed" (which unfortunately has opposite meaning from "modified" but is more accurate in capturing what the flag means).

> Likewise the docstring for RichCollectionBase.modified could be a bit more helpful (modified since when?).
>
> The maxsize of the _put_queue should be a class constant. (And I suppose num_put_threads and num_get_threads might as well be class variables or even class constants, until/unless there's some motivation/mechanism for changing them at runtime.)

Done.

> The "mode" argument to ArvadosFileReader seems superfluous: the only mode that makes sense is "r". Perhaps the argument should be omitted, or (since having the same signature seems nice) it might be worth throwing an exception if mode is not "r", instead of just ignoring it.

Done, the reason it was there in the first place was so that ArvadosFileWriter could pass "mode" up through the __init__ methods to _FileLikeObjectBase, but now ArvadosFileWriter just sets self.mode directly.

> FileWriter and FileReader are documented to be thread-unsafe, but it seems like we could throw self.lock = threading.RLock() and a @synchronized decorator on the methods that use _filepos, and they'd become thread-safe. Is this correct? (This probably shouldn't hold up the branch if it's not as easy as that.)

_FileLikeObjectBase and ArvadosFileReaderBase would also have to be made thread safe, so this isn't totally trivial. Also, FUSE doesn't use ArvadosFileReader and ArvadosFileWriter so I think this is out of scope of this ticket.

> This seems to be a pre-existing bug, but: RichCollectionBase.mkdirs is like os.makedirs() (not os.mkdirs()), but unlike os.makedirs() does not raise an exception if the target already exists. It also returns an ArvadosFile (instead of raising an exception as one would expect) if the target already exists and is a file.

Fixed.

#38 - 06/12/2015 02:11 PM - Peter Amstutz

Tom Clegg wrote:

> I think conflict filenames should avoid using ":" characters. They have a tendency to confuse programs that know about host.name:dir/file syntax. How about just YYYY-MM-DD-HHMMSS? Perhaps we could also use something like "foo~timestamp~conflict~" instead of "foo~conflict-timestamp~" so users/scripts can safely remove "*~conflict~"? (Globs like "*~conflict*~" seem less safe, if only slightly...)

Done.

> I find the test cases hard to follow because "file1", "file2", "collection1", "collection2", "foo", and "bar" don't tell me what role they play in the test scenario. For example, perhaps fuseWriteFileTestHelper1 and fuseWriteFileTestHelper2 could be called fuseWriteHelloWorldToFile1 and fuseReadHelloWorldFromFile1...?

Improved some of the function naming.

> Can we move the fuse*Helper functions inside the test classes they serve? I think that would make it much easier to see how the code fits together.

Unfortunately we can't, the multiprocess module fails if you do that. Something about not being able to pickle class members.

> I don't think this test has any business asserting the exact format of a permission signature, the absence of additional hints, etc. I'd say if it matches ...a20(\+\S+)? 0:12:... then fuse performed correctly.
> [...]

Relaxed the regex.

> Each test case takes about 4 seconds, most of which (I hope!) is setup/teardown. Can we do better? We could easily reduce the 0.5 second granularity in the unmount wait loop, but the setup seems even more expensive. Can we leave the mount alive across several test cases, and use some other way to give each test case a sufficiently blank state? (Although test independence is valuable, it would also be somewhat reassuring to test that the mount keeps working for longer than a single test case.) Perhaps we can also speed up the mounting process?

It's more like 2 seconds for me. Unfortunately a lot of these tests work by creating a Collection object that is mounted as the FUSE root, in order to test that FUSE actions correctly change the underlying collection object; it might be technically possible to switch out the collection object and leave the mount up but probably at the expense of a refactoring that makes already complicated tests even more complicated.

> I'd like to see at least one stress test that could conceivably expose unanticipated race conditions. We could even get an interesting performance data point while we're at it. Perhaps something like "start N processes, each of which does M iterations of {copy file1.txt to a new collection called "test%d", commit, write different stuff, commit, try to read one of the other process's "test%d" collection}".

I'll have to give that one some thought. Meanwhile you can go through the current batch of fixes?

> There's a lot of "Calling fuse_mount" spam in fuse tests. I'm guessing it's because of this. Is this left in intentionally?
> [...]

Fixed.

> Please uncomment or remove:
> [...]

Fixed.

> I think this should be self.events:
> [...]

You're right. Fixed.

> Is this logging.DEBUG line left in intentionally?
> [...]

Fixed.

> Is there some situation where a non-collection object_uuid is in inode_cache? The "object_kind" condition here seems a bit surprising:
> [...]

Yes, ProjectDirectory objects also live in the inode_cache. This logic uses event bus to trigger update, but only for objects that are live in the inode cache. The object_kind condition is a special case for collections that bypasses update() if the portable data hash is the same (which happens a lot when FUSE commits to the API server and then gets an event back notifying it of the change that it just made.)

> This is off-putting at best:
> [...]

Hmm, it doesn't do this for me.

> I tried fixing it by putting self.pool.join() before self.pool=None (as the multiprocessing docs seem to suggest) but that just adds a bunch of these messages before the atexit stuff:
> [...]

It's already calling pool.close() so presumably that calls pool.join().

> Does this mean the processes really aren't exiting cleanly for some reason? The test-case-within-multiprocessing-within-test-case stuff seemed a bit sketchy; perhaps "move helpers into test classes" will actually help Python, not just devs?

Hopefully you saw the comment near the top of test_mount, using multiprocessing for tests isn't for fun, it's the least bad way I could figure out to work around the Python GIL and still have relatively self-contained tests. Unfortunately the multiprocessing brings its own quirks, but so far the tests have been stable for me.

#39 - 06/15/2015 03:38 PM - Peter Amstutz

I used iozone (apt-get install iozone3) for performance testing, this uncovered a few bugs and performance issues. After fixing the obvious things, here's some more general observations about fuse performance, testing on my laptop:

- Single file peak read/write reported by iozone is around 65 MiB/s for a 256MiB file
- iozone is using 4k writes and 128k reads
- When reading or writing multiple files the throughput for each individual file is reduced more or less linearly in the number of files (2 files = 30MiB/s, 4 files = 15 MiB/s)
- There's about a 30% penalty on file updates (50 MiB/s vs 65 MiBs) compared to writing the file initially
- top reports arv-mount pegged at 120% CPU

I did some additional testing with dd

- I created a 2GiB file of random data
- Reading with dd I get 140 MiB/s
- During the read, arv-mount uses about 95% CPU
- Writing a file full of zeros (because writing random data bottlenecks on the RNG and not the actual writes) I get 108 MiB/s

Conclusions:

It's pretty clear that we're starting to run up against the limitations of Python for FUSE. The iozone tests used a 256MiB file which fits in the default arv-mount cache, which means the read/write speeds are bottlenecking entirely on the ability for the Python code to deliver the results and (for concurrent requests) on the GIL.

#40 - 06/15/2015 05:54 PM - Peter Amstutz

A couple additional data points:

- I wrote a small script to iterate over the same file using the Python SDK directly. This downloaded 2 GiB of random data in 11.9s for about 170 MiB/s and the Python process using 80% CPU
- This suggests that FUSE has around 15% overhead compared to using the SDK directly.
- I added up the reported time per request from the Keep server logs and got a total of 5.4s for 39 block requests. This means about 45% of the time is spent in the Keep server and 55% is spent in the client. In this data set, a complete 64 MiB blocks is served in about 0.17s.
- a 64 MiB block in 0.17s means keepstore is performing at 375 MiB/s
- Going into the keep directory and doing time cat 34a45ec404699fcaa9792a9c066f5ce9 >/dev/null yields a time of .017s so that suggests that keepstore is only spending about 10% of its time actually reading data.

This also suggests that we have the headroom to double performance (300+ MiB/s) by reducing the overhead in the client; it's also possible there are performance enhancements that could happen in keepstore.

I'm curious how much of that time is spent in md5. One thing I checked was to see if the md5 module was written in Python, but in fact it is a C module.

We need to do additional profiling to determine where the bottleneck is.

#41 - 06/16/2015 03:15 PM - Tom Clegg

This has gotten rather unwieldy:

```
        if ((self._state == _BufferBlock.WRITABLE and nextstate == _BufferBlock.PENDING) or
            (self._state == _BufferBlock.PENDING and nextstate == _BufferBlock.COMMITTED) or
            (self._state == _BufferBlock.PENDING and nextstate == _BufferBlock.ERROR) or
            (self._state == _BufferBlock.ERROR and nextstate == _BufferBlock.PENDING)):
            # the whole method body
        else
            raise ...
```

I think it's worth making this explicit as a check for permitted state transitions, and outdenting the method body:

```
    STATE_TRANSITIONS = frozenset([
            (WRITABLE, PENDING),
            (PENDING, COMMITTED),
            (PENDING, ERROR),
            (ERROR, PENDING)])

    def set_state(self, nextstate, val=None):
        if (self._state, nextstate) not in self.STATE_TRANSITIONS:
            raise ...

        # method body
```

In locators_and_ranges() in source:sdk/python/arvados/_ranges.py, the docstring says limit is "Maximum segments to return, default None (unlimited)." In that case it seems like the > here should be a >=:

```
     while i < len(data_locators):
+        if limit and len(resp) > limit:
+            break
```

Or, for that matter, since resp can only grow by one element per iteration and len(resp) != None is always going to be true:

```
     while i < len(data_locators) and len(resp) != limit:
```

I'm still getting these. Not every time -- maybe one in 3.

```
Ran 30 tests in 113.093s

OK
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "/usr/lib/python2.7/multiprocessing/util.py", line 284, in _exit_function
    info('process shutting down')
TypeError: 'NoneType' object is not callable
Error in sys.exitfunc:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "/usr/lib/python2.7/multiprocessing/util.py", line 284, in _exit_function
    info('process shutting down')
TypeError: 'NoneType' object is not callable
```

> > tried fixing it by putting self.pool.join() before self.pool=None (as the multiprocessing docs seem to suggest)
>
> It's already calling pool.close() so presumably that calls pool.join().

I think we should join() here in order to eleminate some race conditions. According to my reading of the multiprocessing docs, close and join are distinct operations:

- close(): Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.
- join(): Wait for the worker processes to exit. One must call close() or terminate() before using join().

(That last sentence is pretty close to a guarantee that close() doesn't call join() by itself...)

It seems that what we want is to kill off all workers (now that they've already done everything we need) and avoid any weird side effects from the timing of worker-shutdown relative to main process shutdown. The appropriate recipe for that seems to be:

```
-self.pool.close()
-del self.pool
+self.pool.terminate()
+self.pool.join()
```

(del self.pool doesn't delete anything: it just unbinds the variable, which seems just as useless as self.pool = None here wrt cleaning up processes. Neither forces garbage collection, which would cause an implicit terminate() -- and even if they did, according to the docs even garbage collection doesn't do a join() ... which seems reasonable, since GC would be a rather unfortunate time to start waiting around for other processes to exit.)

There are several "See {note,comment} in FuseWriteFileTest" comments but FuseWriteFileTest itself doesn't have any comments. "What's the secret, Peter?" :)

I haven't found a flag to enable/disable writable (sorry I didn't notice, or fail to notice, this earlier). I'd suggest:

- --read-only and --read-write flags (or something else that lets you be explicit about both).
- default to read-only -- at least until we see how it behaves in the wild for a bit, and update crunch-job to use an explicit readonly flag.
- It should be documented at source:doc/user/tutorials/tutorial-keep-mount.html.textile.liquid. I don't think this has to be extensive but it should probably say something to the effect of "beta" and possibly mention something about the timing of commits and what it means when a "conflict" file appears.

Is this change intentional, in services/fuse/setup.py?

```
       install_requires=[
-         'arvados-python-client>=0.1.20150303143450',
+         'arvados-python-client',
```

For whatever reason, the tests are running faster for me too now, and ruby+postgres are staying pretty busy so I assume much of the test case overhead is "reset database fixtures". The fuse test suite takes ~110s, which seems tolerable.

Thanks for the performance numbers. It's great to have a reference point so if we find "arv-mount is slow" somewhere we know what it's capable of when nothing is going wrong.

> It's pretty clear that we're starting to run up against the limitations of Python for FUSE.

I don't quite follow that. How do we decide the fault can't be in our own Python code?

Any further thoughts on the stress test? It looks to me like there are a lot of careful synchronization bits here that aren't protected by any tests, which makes it dangerous to work on. Additional possibilities:

- "Big" and "small" versions of the same stress test might be worthwhile, if a truly convincing test is too slow. Jenkins can run the "small" version routinely.
- A standalone diagnostic test might be helpful: a stress test I can run in a real arv-mount that creates a new sandbox project and does a bunch of stuff to it. This might be useful in the long term for bisecting "FUSE didn't do what I expected" problems. I suppose Go would be particularly good at this, but Python with multiprocessing should also be quite capable of invoking a bunch of synchronization code, and much easier to customize in the field for troubleshooting purposes.

#42 - 06/16/2015 08:20 PM - Peter Amstutz

Tom Clegg wrote:

> I think it's worth making this explicit as a check for permitted state transitions, and outdenting the method body:

Done.

> Or, for that matter, since resp can only grow by one element per iteration and len(resp) != None is always going to be true:

Done.

> (del self.pool doesn't delete anything: it just unbinds the variable, which seems just as useless as self.pool = None here wrt cleaning up processes. Neither forces garbage collection, which would cause an implicit terminate() -- and even if they did, according to the docs even garbage collection doesn't do a join() ... which seems reasonable, since GC would be a rather unfortunate time to start waiting around for other processes to exit.)

Changed close() to terminate() and added join(). Note that del self.pool actually **does** do something, assuming that's the last reference to self.pool, the reference count should go to zero, and it will be deleted, which includes running finalizers (which I think is where it starts throwing errors due to running finalizers during shutdown).

> There are several "See {note,comment} in FuseWriteFileTest" comments but FuseWriteFileTest itself doesn't have any comments. "What's the secret, Peter?" :)

The note moved, the comments did not. Fixed.

> I haven't found a flag to enable/disable writable (sorry I didn't notice, or fail to notice, this earlier). I'd suggest:
>
> - --read-only and --read-write flags (or something else that lets you be explicit about both).
> - default to read-only -- at least until we see how it behaves in the wild for a bit, and update crunch-job to use an explicit readonly flag.

Done.

> - It should be documented at source:doc/user/tutorials/tutorial-keep-mount.html.textile.liquid. I don't think this has to be extensive but it should probably say something to the effect of "beta" and possibly mention something about the timing of commits and what it means when a "conflict" file appears.

Done.

> Is this change intentional, in services/fuse/setup.py?

Well, the version peg should be to the new python package version that will get published once this merges.

> For whatever reason, the tests are running faster for me too now, and ruby+postgres are staying pretty busy so I assume much of the test case overhead is "reset database fixtures". The fuse test suite takes ~110s, which seems tolerable.
>
> Thanks for the performance numbers. It's great to have a reference point so if we find "arv-mount is slow" somewhere we know what it's capable of when nothing is going wrong.

140 MiB/s (or 60 MiB/s) is not too bad, this is probably fast enough for most applications we want, although loading 70 GiB of data is still going to be in the 10 - 20 minute range. This is under ideal circumstances (SSD, localhost) so benchmarking actual installs is going to be important.

>> It's pretty clear that we're starting to run up against the limitations of Python for FUSE.
>
> I don't quite follow that. How do we decide the fault can't be in our own Python code?

Let me rephrase that. We are running up against the limitations of using Python to implement a FUSE driver. Throughput is not I/O limited, it is CPU limited (I did some profiling already and fixed the obvious hot spots, it's not obvious how to optimize the remaining code), and concurrency is clearly limited by the GIL and llfuse locks (as demonstrated when reading/writing files in separate collections so there isn't collection lock contention).

That said, it might be worth looking into PyPy (although it also has a GIL) but I don't know whether llfuse and our other dependencies will work with it.

> Any further thoughts on the stress test? It looks to me like there are a lot of careful synchronization bits here that aren't protected by any tests, which makes it dangerous to work on. Additional possibilities:
>
> - "Big" and "small" versions of the same stress test might be worthwhile, if a truly convincing test is too slow. Jenkins can run the "small" version routinely.
> - A standalone diagnostic test might be helpful: a stress test I can run in a real arv-mount that creates a new sandbox project and does a bunch of stuff to it. This might be useful in the long term for bisecting "FUSE didn't do what I expected" problems. I suppose Go would be particularly good at this, but Python with multiprocessing should also be quite capable of invoking a bunch of synchronization code, and much easier to customize in the field for troubleshooting purposes.

I still don't know what kind of test you're looking for. Several writers to the same file? Several writers to different files in the same collection? Several writers to different files in different collections? Moving files? All of these above? What specifically are do you want the test to do?

#43 - 06/18/2015 05:53 PM - Peter Amstutz

The file system stress test on the fuse driver starts a set of processes each of which go through several cycles of creating, read, and deleting a set of files.

When a file is deleted, it calls invalidate_entry(). Normally this puts an invalidate request into _notify_queue and goes along its way.

When there is a lot of activity in one directory (such as 20 processes stress testing the system!) *notify_queue can start to get backed up. This happens because the underlying FUSE invalidation notification relies on taking the kernel lock for the inode, but there is lock contention with the other FUSE operations such as getattr() and unlink() which also take the inode lock _before* sending the request to FUSE and don't release it until FUSE has responded.

There's currently a limit of 1000 items in _notify_queue, at which point any further calls to invalidate_entry() will block.

IF the FUSE driver receives a request with the parent inode is locked (such as unlink())
AND the head of _notify_queue is an invalidate message in for the same inode
AND _notify_queue is full

AND the handler calls invalidate_entry()

THEN it will deadlock because invalidate_entry() is blocked (because the queue is full), _notify_queue can't drain (the inode is locked), and the FUSE request that's holding the inode lock can't complete (because it's waiting on invalidate_entry()).

The simplest solution seems to be to make _notify_queue much larger (10000 entries instead of 1000) or unlimited size (so this particular problem doesn't reoccur with 200 processes instead of 20).

#44 - 06/18/2015 08:51 PM - Peter Amstutz

Stress testing program is in arvados/services/fuse/test/fstest.py

#45 - 06/22/2015 03:38 PM - Tom Clegg

Peter Amstutz wrote:

> > (del self.pool doesn't delete anything: it just unbinds the variable, which seems just as useless as self.pool = None here wrt cleaning up processes. Neither forces garbage collection, which would cause an implicit terminate() -- and even if they did, according to the docs even garbage collection doesn't do a join() ... which seems reasonable, since GC would be a rather unfortunate time to start waiting around for other processes to exit.)
>
> Changed close() to terminate() and added join(). Note that del self.pool actually **does** do something, assuming that's the last reference to self.pool, the reference count should go to zero, and it will be deleted, which includes running finalizers (which I think is where it starts throwing errors due to running finalizers during shutdown).

Thanks.

Unfortunately, I still get this:

```
Ran 30 tests in 102.295s

OK
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "/usr/lib/python2.7/multiprocessing/util.py", line 284, in _exit_function
    info('process shutting down')
TypeError: 'NoneType' object is not callable
Error in sys.exitfunc:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "/usr/lib/python2.7/multiprocessing/util.py", line 284, in _exit_function
    info('process shutting down')
TypeError: 'NoneType' object is not callable
```

By the way, in case this is related to the test-case-in-process strategy (but also because it seems it could make the tests easier to read and write) to run the mount, rather than the tests, in a separate process? Or is there some reason that doesn't work? (Aside from the "not callable" stuff, I'm hoping to avoid duplicating thought/effort in a future refactoring exercise if this strategy has already been explored and abandoned.)

> I haven't found a flag to enable/disable writable (sorry I didn't notice, or fail to notice, this earlier). I'd suggest:
>
> - --read-only and --read-write flags (or something else that lets you be explicit about both).
> - default to read-only -- at least until we see how it behaves in the wild for a bit, and update crunch-job to use an explicit readonly flag.

This might be a silly thing to debate at this point but could we have symmetrical flags like "--read-only/read-write" or "--{enable/disable}-write" or "--writable={false,true}"? Since they really do just set/clear a single flag, it seems weird to have different ways of saying them (--readonly / --enable-write).

This looks like it should be debug, not warn:

```
    logger.warn("enable write is %s", args.enable_write)
```

Noticed a buggy sentence in docs. Suggestion:

```
-If multiple clients try to modify the same file in the same collection, this result in a conflict.
+If multiple clients modify the same file in the same collection within a short time interval, this can
 result in a conflict.
```

It might also be worth clarifying what "multiple clients" means (multiple writers on same arv-mount *vs.* multiple arv-mount). But the main thing is that a conflict *might* happen.

The list of things you can do seems a bit weird to me (it sounds like a verbose way to say "everything") but I'm inclined to put aside further editing rather than hold up the actual feature.

> Well, the version peg should be to the new python package version that will get published once this merges.

OK... please don't forget to do that. ;) Fortunately it doesn't have to be the exact published version -- any timestamp newer than the last non-3198 merge to sdk/python should do.

> Let me rephrase that. We are running up against the limitations of using Python to implement a FUSE driver. Throughput is not I/O limited, it is CPU limited (I did some profiling already and fixed the obvious hot spots, it's not obvious how to optimize the remaining code), and concurrency is clearly limited by the GIL and llfuse locks (as demonstrated when reading/writing files in separate collections so there isn't collection lock contention).

OK, I'm skeptical about this as proof that it's Python's fault and not our own, although I'd certainly expect the best Go implementation (for example) to be significantly faster than the best Python implementation -- in that sense optimizing our Python code might not be a good long term investment. In any case, if you have some iozone command lines sitting around, it might be handy to have them pasted here.

> That said, it might be worth looking into PyPy (although it also has a GIL) but I don't know whether llfuse and our other dependencies will work with it.

Indeed, that sounds worth investigating. Especially with that stress test in hand. :P

#46 - 06/25/2015 02:42 PM - Peter Amstutz

Tom Clegg wrote:

> By the way, in case this is related to the test-case-in-process strategy (but also because it seems it could make the tests easier to read and write) to run the mount, rather than the tests, in a separate process? Or is there some reason that doesn't work? (Aside from the "not callable" stuff, I'm hoping to avoid duplicating thought/effort in a future refactoring exercise if this strategy has already been explored and abandoned.)

On further research, the error on exit appears to be a Python 2.7.3 bug: https://bugs.python.org/issue15881 this bug is fixed in Python 2.7.4+ which explains why I don't get the error (I am using Python 2.7.9).

Running the mount, rather than the tests in a separate process would not fix the problem since it would either need to use multiprocessing (and be subject to the same bug) or I would have to hand-roll a forking strategy. Also, several tests rely on initializing FUSE with a specific SDK Collection object and then inspecting the Collection object to validate that the FUSE operation are modifying the underlying object in the expected way.

> This might be a silly thing to debate at this point but could we have symmetrical flags like "--read-only/read-write" or "--{enable/disable}-write" or "--writable={false,true}"? Since they really do just set/clear a single flag, it seems weird to have different ways of saying them (--readonly / --enable-write).

Now --read-only and --read-write.

> This looks like it should be debug, not warn:
> [...]

Fixed. Now info().

> Noticed a buggy sentence in docs. Suggestion:
>
> [...]
>
> It might also be worth clarifying what "multiple clients" means (multiple writers on same arv-mount *vs.* multiple arv-mount). But the main thing is that a conflict *might* happen.

Done.

> The list of things you can do seems a bit weird to me (it sounds like a verbose way to say "everything") but I'm inclined to put aside further editing rather than hold up the actual feature.

Well, there's also a list of things you can't do, but it's better to be explicit about the capabilities and limitations than imply that it can do something it can't.

> > Well, the version peg should be to the new python package version that will get published once this merges.

> OK... please don't forget to do that. ;) Fortunately it doesn't have to be the exact published version -- any timestamp newer than the last non-3198 merge to sdk/python should do.

I'll do the version pin dance where I merge and follow it up with a commit that fixes setup.py and push both at the same time.

> OK, I'm skeptical about this as proof that it's Python's fault and not our own, although I'd certainly expect the best Go implementation (for example) to be significantly faster than the best Python implementation -- in that sense optimizing our Python code might not be a good long term investment. In any case, if you have some iozone command lines sitting around, it might be handy to have them pasted here.

Instead of speculating, let's wait to see what performance profiling says.

#47 - 06/25/2015 05:49 PM - Tom Clegg

Peter Amstutz wrote:

> > OK... please don't forget to do that. ;) Fortunately it doesn't have to be the exact published version -- any timestamp newer than the last non-3198 merge to sdk/python should do.
>
> I'll do the version pin dance where I merge and follow it up with a commit that fixes setup.py and push both at the same time.

I was just pointing out that the "version pin dance" isn't necessary (there's no version pin, just "not older than X") hoping to save you the trouble, but sure, go for it.

> > OK, I'm skeptical about this as proof that it's Python's fault and not our own, although I'd certainly expect the best Go implementation (for example) to be significantly faster than the best Python implementation -- in that sense optimizing our Python code might not be a good long term investment. In any case, if you have some iozone command lines sitting around, it might be handy to have them pasted here.
>
> Instead of speculating, let's wait to see what performance profiling says.

Exactly. So... any iozone command lines sitting around that you can paste here?

The rest of the changes look good, please merge. Thanks.

#48 - 06/25/2015 05:55 PM - Peter Amstutz

- Status changed from In Progress to Resolved

- % Done changed from 88 to 100

Applied in changeset arvados|commit:5b3187552676947ee74e4b652e7a04d3d9b9a3a4.

#49 - 06/25/2015 06:01 PM - Peter Amstutz

Tom Clegg wrote:

> > Instead of speculating, let's wait to see what performance profiling says.
>
> Exactly. So... any iozone command lines sitting around that you can paste here?

Single file, single process, writing a 512 MiB file:

iozone -s512m -l1 -u1

Four processes each writing a separate 128 MiB file:

iozone -s128m -l4 -u4

> The rest of the changes look good, please merge. Thanks.

Whoohoo!