Item 331
git_comments:

git_commits:

1. **summary:** HDDS-3118. Possible deadlock in LockManager. (#627)
   message: HDDS-3118. Possible deadlock in LockManager. (#627)

github_issues:

github_issues_comments:

github_pulls:

1. **title:** HDDS-3118. Possible deadlock in LockManager.
   body: ## What changes were proposed in this pull request? Possible deadlock in lock manager. This is because of lockPool.borrowObject waits indefinitely and activeLocks.compute is an atomic operation that holds a map lock. ## What is the link to the Apache JIRA https://issues.apache.org/jira/browse/HDDS-3118 ## How was this patch tested? The reproed patch is now passing.

github_pulls_comments:

1. Thanks @bharatviswa504 for the fix and @xiaoyuyao for the review.

github_pulls_reviews:

1. **body:** Nit: Is this necessary? I don't see any checked exception being thrown in the `try` block, and unchecked exceptions don't need to be wrapped.
   label: code-design
2. **body:** Thanks @bharatviswa504 for the patch. The PR LGTM. I understand we are trying to avoid wait forever while holding the lock of the pool map. However, it does not completely solve the lock resource exhausting problem. I have a cluster with this patch but create key still can live lock by the limited number of locks (100). My question is why we are restricting # of locks here via hdds.lock.max.concurrency. In my opinion, if we change the default from 100 to -1, this deadlock will not happen. This also match the default max pool size (-1) of GenericObjectPool, especially lock is a cheaper resource compared with threads or tcp connections. To allow handling 4K+ request per second, we should consider removing this key that can easily cause deadlock, especially when handler counter is several times larger than the lock pool size.
   label: code-design
3. **body:** Yes, if we set max pool size to -1, the max total objects created by GenericcObjectPool used in Integer.MAX. I agree we can change to -1, and we shall not see this issue. And we can remove this property, and set it to -1 so that we shall not see this kind of issue when someone mistakenly sets this to a low value and operates on the cluster with very large number of volumes/buckets in parallel. With this way, we shall not see lock contention during acquiring lock. But few questions, right now we take a lock on resource name, so if there is a lock Object which is already in activeLocks map, we reuse that object if it does not we borrowObject from lockPool. So, when we operate on more than 100 buckets simultaneously we should see this problem when creating keys or when more than 100 volumes operations are going in parallel we can see some contention in acquiring locks. So, it is not exactly related to handler count in OM. Just trying to understand is the issue is observed on the test cluster, where some test is working in parallel across more than 100 volumes/100 buckets and a lock contention is observed?
   label: code-design
4. Thank You @xiaoyuyao for the review. Removed the config in the latest commit.
5. Based on Xiaoyu's comment reverted this change.
6. **body:** NIT: we don't need to explicitly set -1 as the default maxTotal from GenericObjectPool is -1.
   label: code-design
7. yes, I did this expicitly, if in future releases if they changed the defaults it might break when we upgrade the commons-pool2.
8. **body:** bq. Just trying to understand is the issue is observed on the test cluster, where some test is working in parallel across more than 100 volumes/100 buckets and a lock contention is observed? I hit the

problem last week and had been reviewing related code along with the OM stack dumps this week. Found the fix here yesterday, tried it and found that here may not completely solve the problem. Here is the root cause: OMKeyCreateRequest.validateAndUpdateCache requires two locks from the lock pool to finish a key creation. One write lock for the BUCKET at the beginning and one lock for longest prefix path close to the end during prepareKeyInfo call. If you have 200 threads handlers but only 100 lock limit from the lock manager. When the first 100 key creation threads all grab the bucket lock, OM will stuck there forever as we don't have any locks from the pool for the second call. My initial thought is to add a timeout but later found removing the limit seems to be a better solution as they will quickly finish and return the lock without the limit.
   label: code-design
9. Sounds good. +1 pending CI.

jira_issues:

1. **summary:** Possible deadlock in LockManager
   description: {{LockManager}} has a possible deadlock. # Number of locks is limited by using a {{GenericObjectPool}}. If N locks are already acquired, new requestors need to wait. This wait in {{getLockForLocking}} happens in a callback executed from {{ConcurrentHashMap#compute}} while holding a lock on a map entry. # While releasing a lock, {{decrementActiveLockCount}} implicitly requires a lock on an entry in {{ConcurrentHashMap}}.

jira_issues_comments: