HBASE-28196 Yield SCP and TRSP when they are blocked

Background

This is a long story in HBase. It became a problem when prof-v2 was born and we implemented RS crash recovery and region assignment with it.
The first problem is about updating meta. We need to update meta when doing region assignment, and usually this is one of the steps in TRSP. So if the meta region is not online, the procedure will hang there and retry for a long time. But if the meta region is not online, usually we need to run SCP and another TRSP to bring it online, these procedures also need to consume the workers in the procedure executor. So if all the workers in procedure executor have already been blocked on updating meta, we are in trouble. The current solution is to keep adding new workers if there is no free worker and no progress on all procedures in the procedure executor, and hope finally the procedures which can bring meta region online can be executed.
The second problem is about region state node locking. For keeping the state consistent, we will hold the region state node lock while updating the meta region when changing the region state in TRSP. But in SCP, when we want to schedule a new TRSP or interrupt the current TRSP, we need to hold the region state node lock too. If a TRSP if blocked while changing region state, it could also blocked another SCP.

Basic Idea

For the first problem, I think it is straightforward. We have async client implementation in HBase, so we could use async client when updating meta. In general, we will call the async method to update meta, get the CompletableFuture and store it as a field in the procedure, and then throw a ProcedureSuspendExcetion, to yield the current procedure. And we also need to add a callback to the CompletableFuture, when the operation completes, we should wake up the procedure and add it back to the procedure executor.
The second problem is more complicated, as by design, when acquiring a lock, the thread should be blocked there. And if we yield and then get back, the thread which executes us may be changed, and java's Lock implementation does not support unlock by another thread.
So first, we need to have a simple lock implementation which supports unlock by another thread. This is not very hard, just like a semaphore.
And for the problem itself, a simple way to deal with this is to use 'tryLock' instead of 'lock', if we can not get the lock, we throw a ProcedureSuspendException and retry later. The disadvantage for this solution is performance and fairness. For example, at least we should retry after several tens or hundreds milliseconds, but the lock may have already been released, but we can not run

immediately. And while we are sleeping, another thread may acquire the lock before us and cause us to fail again when retrying.
Another way is more complicated. We can maintain a queue in the region state node, while a procedure can not get the lock, we add it to the queue and throw ProcedureSuspendException. When others release the lock, we populate the queue and wake up the procedures. The challenge here is how to achieve this programmingly, as the procedure will not be blocked in the method, how can we know it is woken up by us to get the lock or it is just another lock call…

Implementation

RegionStateNodeLock

void lock();
void unlock();
boolean tryLock();

These 3 methods are for thread, where we will set the owner as the current thread.

void lock(Procedure<?> proc, Runnable wakeUp) throws ProcedureSuspendException;
void unlock(Procedure<?> proc);
boolean tryLock(Procedure<?> proc);

These 3 methods are for procedure, where we will set the owner as the procedure. Notice that, for locking by procedure, if we can not acquire the lock immediately, we will throw a ProcedureSuspendException, to halt the procedure. The run method for wakeUp will be called when someone else releases the lock and it is the procedure's turn to get the lock. Usually, the wakeUp is just a wrapper of adding the procedure back to ProcedureScheduler.

As said above, we need to deal with the problem on how to let the procedure which has been woken up to get the lock, so we introduced another method

boolean isLockedBy(Object lockBy);

In the implementation of RegionStateNodeLock, before waking up a procedure, we will set the lock's owner to this procedure. And in the procedure's execute method, before calling the lock method, we should call isLockedBy method to check whether we have already held the lock, if so, we do not need to hold it again.

ProcedureFutureUtil

Introduce a general util for implementation suspend/wake when doing asynchronous operation in a procedure.
In general, the procedure should have a field for storing a CompletableFuture.

When the field is present, we will reset it to null and then call the get method for this CompletableFuture, to get the result of the operation. Usually this will not be a blocking operation as we will only be woken up if the CompletableFuture is finished.
When the field is null, we will do the asynchronous operation and get the CompletableFuture. If the CompletableFuture is already finished, we just go on. Otherwise, we store the CompletableFuture to the field, and also add a callback to it, to wake up the procedure when the future is complete, and then throw a ProcedureSuspendException to suspend the procedure.

Potential deadlock problem

The implementation for RegionStateNodeLock is reentrant, but if you have already called lock(Procedure<?> proc), calling lock() in the same thread will lead to a deadlock and make you hang there forever. Developers should try to avoid these usages when writing code.