Application Locks (or Mutexes) in SQL Server 2005

Written by **Mladen Prajdić** on **07 January 2008** | **0 Comments**
http://www.sqlteam.com/article/application-locks-or-mutexes-in-sql-server-2005

Application locks aren't a well known area of locking in SQL Server, but they can be very useful for special scenarios. They work in a analogous way to the `lock()` construct in .Net and are basicaly user defined mutexes in SQL Server.

An application lock is a bit different than other kinds of SQL Server locks though. While other locks lock schema or data, application locks lock a part of your code. There are 2 stored procedure that are used for this: `sp_getapplock` and `sp_releaseapplock`.

```
USE AdventureWorks;

BEGIN TRANSACTION
    DECLARE @res INT
    EXEC @res = sp_getapplock
                -- unique name nvarchar(255), truncated to 255 if
longer
                @Resource = 'This a Lock ID',
                -- Valid values: Shared, Update, IntentShared,
IntentExclusive, Exclusive
                @LockMode = 'Exclusive',
                -- Scope of the lock: Transaction or Session
                @LockOwner = 'Transaction',
                -- Timeout in miliseconds
                @LockTimeout = 60000,
                -- db principal that has access permisions
                @DbPrincipal = 'public'

    -- we can see our lock in in the DMV with resource_type =
'APPLICATION'
SELECT resource_type, request_mode,
     resource_description
FROM   sys.dm_tran_locks

    -- 0 and 1 are valid return values
    IF @res NOT IN (0, 1)
    BEGIN
        RAISERROR ( 'Unable to acquire Lock', 16, 1 )
    END
    ELSE
    BEGIN
        SELECT * FROM Person.Address
        EXEC @res = sp_releaseapplock
```

```
                        @Resource = 'This a Lock ID'
@DbPrincipal = 'public',
                        @LockOwner = 'Transaction'
    END
COMMIT
```

If an application lock owner is a transaction, the lock gets automatically released when the transaction ends.

However to be able to call `sp_getapplock` a user calling the stored procedure must meet one of these conditions:

- is dbo
- is in the db_owner role
- is the DB Principal ID (e.g. guest)
- is in the DB Principal ID role (e.g. public)

Best understood with an example...

Note that application locks aren't taken on any data like standard locks. Let's illustrate with some code derived from above code. We need 3 batches:

Batch 1

```
BEGIN TRAN
EXEC @res = sp_getapplock
....
SELECT *
FROM Person.Address
```

Batch 2

```
BEGIN TRAN
EXEC @res = sp_getapplock
....
SELECT *
FROM Person.Address
```

Batch 3

```
SELECT *
FROM
Person.Address
```

Run batch 1: Begin a transaction, get an application lock and select data from Person.Address, but don't release the lock nor end the transaction.

Run batch 2: Begin a transaction, which will try to get an application lock but it won't be able to since the application lock with the same name (@Resource) already exists. The batch will wait until the lock with the existing name (@Resource) is released or the transaction is ended which automatically releases the application lock.

Run batch 3: This will always run disregarding the application lock altogether since there are no real locks on data.

... and of course with an another example

A great example of application locks is a typical business logic problem of inserting data if it doesn't exist and update it if it does. I've written about this in this **blog post**. In it I looked at the locking being held and the post comments have great value. But however you try to make this work you'll always run into some concurrency issues. If you put the whole thing into a transaction then you’ll get into situations with violating PK constraint then inserting data. Another option is to use XLOCK and HOLDLOCK hints in a transaction but this can result in a deadlock which is even worse that the first situation. Application locks prove to be a very good solution to this:

```
USE tempdb
GO

IF OBJECT_ID('AppLockTest') IS NOT NULL
    DROP TABLE AppLockTest
IF OBJECT_ID('spTestAppLocks') IS NOT NULL
    DROP PROC spTestAppLocks
GO

CREATE TABLE AppLockTest( id INT, val VARCHAR(10))
INSERT INTO AppLockTest
SELECT 1, 'value 1' UNION ALL
SELECT 2, 'value 2'
GO

CREATE PROC spTestAppLocks
    @id INT,
    @val VARCHAR(10)
AS
BEGIN TRANSACTION
    DECLARE @res INT
    EXEC @res = sp_getapplock
                @Resource = 'Upsert_app_lock',
                @LockMode = 'Exclusive',
                @LockOwner = 'Transaction',
                @LockTimeout = 60000,
                @DbPrincipal = 'public'
    PRINT 'LOCK ACQUIRED: start our upsert'
    -- 0 and 1 are valid return values
    IF @res NOT IN (0, 1)
    BEGIN
        RAISERROR ( 'Unable to acquire Lock', 16, 1 )
    END
    ELSE
    BEGIN
        -- just to see that the stored procedure will wait
        -- for the completion of the previous one.
        WAITFOR DELAY '00:00:10'

        IF EXISTS (SELECT * FROM AppLockTest WHERE id = @id)
        BEGIN
            UPDATE AppLockTest
```

```
                SET val = @val
                WHERE id = @id
            END
            ELSE
            BEGIN
                INSERT AppLockTest
                SELECT @id, @val
            END
            EXEC @res = sp_releaseapplock
                                    @Resource = 'Upsert_app_lock',
                                    @DbPrincipal = 'public',
                                    @LockOwner = 'Transaction'
            PRINT 'LOCK RELEASED: end our upsert'
        END
COMMIT
GO

-- Run in Query Window 1
EXEC spTestAppLocks 1, 'val 1a'

-- Run in Query Window 2
EXEC spTestAppLocks 1, 'val 1b'
```

If you run the stored procedure in Query Window 1 and after 5 second your run the stored procedure in Query Window 2 you'll see that the whole code between `sp_getapplock` and `sp_releaseapplock` won't execute until the stored procedure in Query Window 1 finishes.

If all of your update/insert logic follows uses this pattern then you'll never get concurrency issues. Of course this method doesn't apply to all environments, so if you can use it requires some testing and design considerations.

Conclusion

While not often used they can come in handy in complex business logic cases. You can see that an application lock simply locks the part of your T-SQL code and not actual data. So to achieve mutual exclusion (mutex), all access has to follow this same lock acquisition pattern using `sp_getapplock` and `sp_releaseapplock`. This is of course best achieved with stored procedure which is another plus in favor of them over ad-hoc (parameterized) queries.