

Play Nice with Others: Waiting for a Lock on SAS Table

John Leveille, d-Wise Technologies, Inc., Raleigh, NC
 Jimmy Hunnings, d-Wise Technologies, Inc., Raleigh, NC

ABSTRACT

It is an all too familiar occurrence for people who run batch jobs during the off hours: You come to work in the morning to find that the overnight jobs failed. Some person or program locked the tables that your program needed and you paid the price -- another night of processing down the tubes. There isn't much that your program can do if the tables you need are locked for the whole night, but what about those times where it was just unfortunate timing and the other person or program was only using the tables for a short time? If your job had just been patient instead of failing right away it would have gotten the data it needed.

This paper demonstrates how to use the `LOCK` statement in base SAS® to cause your program to wait until the tables you need are available. The techniques shown can be used to protect critical sections of code by locking multiple tables. Applying these concepts can greatly improve the reliability of programs that run in highly contentious batch environments.

INTRODUCTION

In many cases, SAS is used to massage data in preparation for analysis and reporting. This usually means that the data used by your SAS programs are copied from some original data source. These data sets exist in a private location that no other programs will access. In this scenario, you don't have to worry about locking your SAS tables or dodging other programs.

In other cases you want to control data sets during parallel processing or control updates to operational data sources. SAS provides the `LOCK` statement to help you do this. The lock statement is a SAS statement that locks a library or a single table until you free the lock or end your SAS session. The syntax for the lock statement is

```
LOCK libref<.member-name<.member-type | .entry-name.entry-type>> <LIST | QUERY  
| SHOW | CLEAR> ;
```

Issuing the statement is quite simple and very similar to issuing a `libname` statement. The statement syntax shown here can be used for local SAS data sets and for data sets served by a SAS/Share® server.

SHARE WITH YOUR FRIENDS

One of the first things children are taught in preschool is to share with others. When a child resists the urge to hog all of the toys he/she is praised for taking the high road. At the office, it only takes a few data set contention errors before you conclude that programmers who share should be commended, too.

At first glance it would seem that the SAS lock statement will easily solve any contention in your SAS programs. However, after you use it a few times, you will realize that the lock statement all by itself is not a complete solution.

When you issue the lock statement and the data set is available for locking you will see a note in the SAS log like this:

```
9 lock mylib.mydata;  
NOTE: MYLIB.MYDATA.DATA is now locked for exclusive access by you.
```

If the same lock statement is submitted but the data set is unavailable for locking you will see a message like this:

```
3 lock mylib.mydata;
```

ERROR: A lock is not available for MYLIB.MYDATA.DATA, lock held by another process.

If you are running interactive code then you can visually inspect the log and determine if the lock was successful. If your program is running in batch then you can check the macro variable SYSLCKRC to determine if the lock was successful. If the lock was not successful then you may want to exit the batch program with an abort statement because your program is probably going to fail.

Unfortunately, exiting a batch program when the lock is unavailable yields nearly the same result as before. With the lock statement in place you will fail gracefully instead of abruptly, but that isn't really succeeding, now is it? If you want to improve the success rate of your batch program what you really need is some logic that will wait until it can lock the necessary tables.

The `trylock` macro combines the lock statement with the necessary logic to create a complete locking solution. Here is the source code for the macro:

```
%macro trylock(member=,timeout=10,retry=1);
  %local starttime;
  %let starttime = %sysfunc(datetime());

  %do %until(&syslckrc <= 0
    or %sysevalf(%sysfunc(datetime()) > (&starttime + &timeout));
    %put trying open ...;

    data _null_;
      dsid = 0;
      do until (dsid > 0 or datetime() > (&starttime + &timeout));
        dsid = open("&member");
        if (dsid = 0) then rc = sleep(&retry);
      end;
      if (dsid > 0) then rc = close(dsid);
    run;

    %put trying lock ...;
    lock &member;
    %put syslckrc=&syslckrc;
  %end;
%mend trylock;
```

The `trylock` macro begins with the `%macro` statement, and it accepts up to three named parameters as input. The first parameter is `member`, and that should be the name of the dataset you want to lock. The second parameter is `timeout` which is the number of seconds that you wish to keep trying to get the lock. The default timeout is ten seconds. If you do not supply a value for this macro parameter then the macro will assign the default value for the timeout. The last parameter is `retry`. This is the amount of time that the macro should wait in between each attempt to lock the table. Inside the macro, the `sleep` function is used to create a time delay before the macro checks again for the availability of a lock. Please note that the `sleep` function is a host-specific function. On Windows hosts the `sleep` function accepts a number of seconds. On Unix hosts the `sleep` function accepts a time value in milliseconds. So if you want the `trylock` macro to sleep for 1 second between lock attempts you should pass `retry=1` on a Windows host and `retry=1000` on a Unix host.

To call the macro on the dataset `mylib.data1` using the default timeout and retry values, you would use the following statement:

```
%trylock(member=mylib.data1)
```

If, instead, you want to try for the lock every 5 minutes for an hour, you would use the following statement:

```
%trylock(member=mylib.data1, timeout=3600, retry=300)
```

The first couple of lines in the macro declare a local variable called `starttime` and assign it the value of the current date and time in seconds. This variable will serve as the baseline for comparisons to determine when the macro should timeout.

```
%local starttime;
%let starttime = %sysfunc(datetime());
```

The next few lines begin the outer loop which will continue to execute until the lock is successful or the macro times out. Since it is a do until statement, the exit conditions will not actually be checked until the first iteration of the code inside the loop has completed. However, we will go ahead and discuss the exit conditions now.

The first condition checked is if the macro variable `syslckrc` is less than or equal to zero. The macro variable `syslckrc` is the return code from the last lock statement. A zero value indicates a successful lock and a less than zero value indicates that you already have a lock on the dataset. The second condition checked is if the macro should timeout. This will be true when the current time is greater than the end time at which the macro should timeout. The current time is represented by the `%sysfunc(datetime())` portion. The end time of the macro is represented as the `starttime` plus the timeout value. You have to use `%sysevalf` due to the fact that a `datetime` value is an integer with many digits causing SAS macro to treat it as a character and refuse to do the math.

```
%do %until(&syslckrc <= 0
           or %sysevalf(%sysfunc(datetime()) >
                       (&starttime + &timeout)
                      )
          );
```

The outer loop contains a SAS `data _null_ step`. The DATA step is used to try to open the data set that was passed in via the `member` parameter. If the attempt to open the data set succeeds then it is available to be locked. Why not just try to lock it with the lock statement? The answer is because the lock statement generates a line of output every time it is invoked. This can easily generate a lot of output lines that will clutter up your SAS log file.

Inside the data step, the variable `dsid` (short for data set identifier) is assigned to zero. When a data set is successfully opened it returns a value greater than zero for its dataset identifier. Next, the DATA step enters a loop alternately sleeping and trying to open the data set. This continues until the data set is successfully opened or the timeout period expires. The last thing the DATA step will do clean up by closing the dataset if it was opened.

```
data _null_;
  dsid = 0;
  do until (dsid > 0 or datetime() > (&starttime + &timeout));
    dsid = open("&member");
    if (dsid = 0) then rc = sleep(&retry);
  end;
  if (dsid > 0) then rc = close(dsid);
run;
```

The next statement warranting discussion is the long awaited lock statement itself. It attempts to lock the data set that was passed into the macro. Since the DATA step did some preliminary checking, it is likely that the lock statement will now succeed. However, due to split-second timing it is still possible that the lock statement will fail. If the lock is successful the outer loop and the macro will terminate. Otherwise, the outer loop will continue looping until the lock is successful or the timeout occurs.

It is important for you to remember to clear the lock on a data set when your job has finished processing so that other users' jobs may use the dataset. You can clear the lock by issuing the lock statement with the `clear` option like this:

```
lock mylib.data1 clear;
```

TAKE A NUMBER

One specific application where locking is particularly useful is the concept of a sequence value. In SAS this is often done by using a 1 row, 1 column data set that holds a numeric value. This value stores the next unique key value that will be used as a primary key in another data set. A program that is filling a table with data opens the sequence table, reads the number stored there, increments the number, and stores it back in the sequence table. This type of program implicitly assumes that two programs are not using the sequence at the same time. If they are then two different programs can acquire the same "unique" key and the results can be very problematic.

In order to prevent the assignment of multiple unique keys, simply preface this program with a trylock macro call and end with a lock statement to clear the lock. The transactions against the sequence table should be very quick so a short timeout and retry period should work well in most cases. For example:

```
%trylock(member=mylib.myseq, timeout=5)

proc sql noprint;
select keyval + 1 into :seqval from mylib.myseq;
update mylib.myseq set keyval = keyval + 1;
quit;

%put Got next key value: &seqval;

lock mylib.myseq clear;
```

CONCLUSION

By using the macro provided in this paper (or similar programming utilizing the same strategy), you can limit the number of mornings that you come into work to face the daunting task of rerunning some or all of your batch jobs. In a highly contentious batch environment, use of a simple locking macro can help you get your work done and still play nice with the other programmers.

REFERENCES

SAS Institute Inc. (2004), SAS 9.1.3 Language Reference: Dictionary, Volumes 1 and 2, SAS OnlineDoc 9.1.3, Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

John Leveille
d-Wise Technologies, Inc.
3115 Belspring Ln
Raleigh, NC 27612
Voice: 919.880.9068
Fax: 919.510.8391
jleveille@d-wise.com
<http://www.d-wise.com>

Jimmy Hunnings
d-Wise Technologies, Inc.
3115 Belspring Ln
Raleigh, NC 27612
Voice: 919.413.5607
Fax: 919.510.8391
jhunings@d-wise.com
<http://www.d-wise.com>