Paper 3380-2015

From a One-Horse to a One-Stoplight Town: A Base SAS® Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks

Troy Martin Hughes

ABSTRACT

Data access collisions occur when two or more processes attempt to gain concurrent access to a single data set, and represent a common obstacle to SAS practitioners in multi-user environments. As SAS® instances expand to infrastructures and ultimately empires, the inherent increased complexities must be matched with commensurately higher code quality standards[i]. Moreover, permanent data sets will attract increasingly more devoted users and automated processes clamoring for attention. As these dependencies increase, so too does the likelihood of access collisions that, if unchecked or unmitigated, leads to certain process failure. The SAS/SHARE® module[ii] offers concurrent file access capabilities, but causes a (sometimes dramatic) reduction in processing speed, must be licensed and purchased separately from Base SAS®, and is not a viable solution for many organizations. Previously proposed solutions in Base SAS® employ a busy-waiting spinlock cycle to repeatedly attempt file access until process success or timeout. While effective, these solutions are inefficient because they generate only read-write locked data sets that unnecessarily prohibit access by subsequent read-only requests. This text introduces the %LOCKITDOWN macro that advances previous solutions by affording both read-write and read-only lock testing and deployment. Moreover, recognizing the responsibility for automated data processes to be reliable, robust, and fault tolerant, %LOCKITDOWN is demonstrated in the context of a macro-based exception handling paradigm.

INTRODUCTION

Gustine, California: population 5,500; traffic lights 0. I grew up in a small town in which unattended, four-way intersections are not uncommon; the population density and paucity of vehicles simply doesn't warrant stoplights. Although hitting a sheep or cow still remains a larger risk to motorists today than other vehicles, as the population of Gustine has increased over the years, so too has the implementation of yield and stop signs, to the near extinction of our sign-less, self-governed intersections. Nevertheless, in its 99 year history, Gustine proudly has sufficed without stoplights. Larger cities, however, require infrastructure and automated controls that prevent gridlock and collisions, as well as prescribe right-of-way and facilitate an efficient flow of traffic. Similar coordination and efficiency are essential to the design of robust data infrastructures and, to that end, this paper describes the implementation of a stoplight that can be applied systematically to data process flows, the %LOCKITDOWN macro.

Like the developing infrastructure of a small town, complex data infrastructures often evolve from humble beginnings in an accretionary or even accidental fashion. Initial processes may be sparse, independent, run infrequently, and require little to no coordination for success. Over time, as these disparate processes become more standardized and static, they are more likely to be serialized into dependent data flows and automated through the operating system (OS) or SAS scheduler. As data processes become more reliable, the use, popularity, and expectation of availability of their data products correspondingly increase. Thus, as the complexity, density, throughput, and interdependency of data processes increase, so too does the need for data access and flow controls. Because Base SAS does not permit concurrent access to a data set while it is being modified, and because access failure causes runtime errors, any process attempting to access a permanent data set should first test its availability. If a data set is unavailable, the process should enter a busy-waiting spinlock in which access repeatedly is tested until the data become available.

But what happens if a data set remains locked indefinitely or for an unacceptably long time? This also occurs when you're driving to In-N-Out Burger® and the stoplight turns red in front of you. Under *normal* circumstances, after a few minutes, the traffic passes, your light turns green, and you proceed. But this time, under *exceptional* circumstances—for example, an endless string of traffic or a train passing before

you—the light never turns green and you eventually make a U-turn. Do you immediately drive thirty miles out of your way to a different In-N-Out Burger® for that Double-Double® and tasty french fries? Do you retrace your steps the following day and reattempt the first restaurant? Or, do you concede defeat and simply head home to make an unsatisfying grilled cheese sandwich?

In software development, similar perils and tough decisions are processed through event handling and exception handling routines in a prescribed, predetermined manner. Thus, robust code often reads like a Twistaplot® or Pick-a-Path® novel of the 1980s[iii], as events (e.g., locked data sets) and exceptions (e.g., *indefinitely* locked data sets) are identified and handled. The goal of exception handling is to facilitate robust software that identifies problems and reroutes processes to favorable end-states or, if all else fails, to graceful program termination. Failure to identify events and exceptions and to handle them responsibly leads to certain process failure. For example, in your quest for In-N-Out Burger®, if no stoplight exists, you might drive directly into traffic or that train in a failure to recognize an event—the congestion. Or, if you fail to recognize the light is never turning green, you potentially could starve to death waiting for the red light to change because you've established no threshold for when to give up. Thus, both event and exception handling are required to facilitate robust, reliable program execution.

The %LOCKITDOWN macro represents a dynamic solution that can be applied with little overhead to Windows- and UNIX-based processes that access communal data sets stored in permanent SAS libraries. Its implementation provides an efficient solution to an obstacle prevalent in multi-user Base SAS environments. Furthermore, by eliminating concern about data access collisions, it facilitates a paradigm shift from antediluvian, serialized data processing to more efficient concurrent processing models.

LOCK BASICS

Even in Base SAS, concurrent data access does not always imply or produce an error; a collision only occurs if one process is attempting to modify a data set while another is attempting to access the same data set. For example, SAS allows two processes concurrently to read the same data set. Throughout this text, data "availability" reflects data sets that both exist and which the requested lock can be obtained. Because a single user can spawn multiple SAS sessions, "multi-user environment" references either multiple users on the same SAS network or a single user running multiple, simultaneous instances of SAS. Thus, even for a single user operating SAS on a non-networked laptop, file access collisions can cause process failure as well as limit the potential and creativity of SAS process flows that can be implemented.

SAS differentiates shared and exclusive file locks[iv]. An exclusive lock (i.e., read-write) is required when a data set is being created or modified and ensures that no other process can access or modify the data; the data are assumed to be in flux and thus cannot even be viewed. A shared lock (i.e., read-only) can access a data set but cannot alter it; its intent is to ensure that no other processes modify the data set while it is being viewed or otherwise utilized. An exclusive lock can be obtained only by a single process and user, whereas multiple processes from multiple users can concurrently obtain shared locks on the same data set. If one or more shared locks are maintained on a data set, an exclusive lock cannot be obtained until all shared locks are released. Although locks can be generated at the library, data set (i.e., member), and observation levels[v], only member-level locking is discussed and implied throughout this text.

Most locks are obtained implicitly by invoking a SAS procedure or DATA step and are released automatically upon DATA step or procedure termination. The type of lock obtained (i.e., degree of exclusivity) is determined by whether the operation will permit the data set to be modified. Input/output (I/O) functions such as OPEN or FOPEN also create implicit locks. Invoking OPEN or FOPEN through the %SYSFUNC macro function creates an implicit lock but, because a data step is not required, this method requires a subsequent %SYSFUNC(CLOSE) statement to clear the lock when the data are no longer in use. Failure to execute this CLOSE function will result in the SAS session unnecessarily maintaining the read-only lock until the session is terminated.

In contrast to *implicit* locks, the LOCK statement generates an *explicit* lock that remains after program termination and until being manually reset through a LOCK CLEAR statement. Whereas a LOCK statement returns a runtime error if a data set is unavailable, the above I/O functions produce return codes that indicate the lock could not be obtained, thus they are preferred in certain circumstances such

as exception handling routines. In the %LOCKITDOWN macro, the FOPEN function, %SYSFUNC(OPEN) function, and LOCK statement each are utilized to provide full functionality.

Table 1 represents an abbreviated list of DATA steps, procedures, functions, and statements and their respective lock types and lock release mechanisms. In each example, the lock type references the lock obtained by the data set test.locked. A more exhaustive list demonstrating lock types is located in SAS/SHARE documentation.[vi]

Code	Lock Type	Release Type
`LOCK test.locked`	exclusive	manual, with LOCK CLEAR
`DATA test.locked;` `  blah blah blah;`	exclusive	automatic, after RUN
`DATA xyz;` `  SET test.locked;`	shared	automatic, after RUN
`PROC SQL;` `  CREATE test.locked`	exclusive	automatic, after RUN

`PROC SQL;`
`  CREATE xyz`
`  FROM test.locked`
	shared	automatic, after RUN

`PROC SORT DATA=test.locked;`	exclusive	automatic, after RUN
`PROC SORT DATA=test.locked OUT=xyz;`	shared	automatic, after RUN
`PROC PRINT DATA=test.locked;`	shared	automatic, after RUN
`DATA _null_;` `  OPEN("test.locked");`	shared	automatic, after RUN

`%let`
`loc=%SYSFUNC(PATHNAME(test))\locked.sas7bdat;`
`FILENAME myfile "&loc";`
`DATA _null_;`
`  f=FOPEN('myfile','U');`
	exclusive	automatic, after RUN

`%let dsid=%sysfunc(open(test.locked));` `%let dsid_close=%sysfunc(close(&dsid));`	shared	manual, with %SYSFUNC(CLOSE)

Table 1. Lock Types and Releases of DATA Steps Procedures, Functions, and Statements

Data access collisions can be mitigated in part through various techniques, one of the simplest being to limit the amount of time that each process requires. By reducing runtime through efficiency techniques such as the use of arrays, hash joins, indices, or the MODIFY statement in a DATA step, a process not only will perform more efficiently but also will lessen the likelihood that other processes simultaneously will request concurrent access to the same data set. Notwithstanding, even the most efficient code—for example, two processes lasting a fraction of a second—eventually will collide and fail if they are attempting to access the same data exclusively. Thus, in a production-grade environment that requires robust, reliable, fault-tolerant processes, locks must be tested and deployed to ensure process success.

LOCK STATEMENT AND THE &SYSLCKRC RETURN CODE

Fundamental to data set availability and lock testing is the eponymous LOCK statement that executes an exclusive lock on a data set so it can be modified. Although the LOCK statement is described in Base SAS reference materials[vii], some options inexplicably function only in a single-session Base SAS environment. Because these idiosyncrasies are not explicitly documented in SAS literature, and because LOCK options performed in a multi-user environment return invalid results, an explanation of single-session and multi-user lock performance and errors is warranted.

In a single-session Base SAS environment, the LOCK LIST statement accurately describes whether an explicit lock has been gained through invocation of the LOCK statement *in that same session*. The

automatic macro variable &SYSLCKRC moreover describes whether the previous LOCK statement was able to achieve a lock or if it produced an error. In the following example, the value of the &SYSLCKRC is zero both after the successful LOCK implementation and the subsequent LOCK CLEAR statement:

```
LOCK test.parent;
NOTE: TEST.PARENT.DATA is now locked for exclusive access by you.
%put &SYSLCKRC;
0
LOCK test.parent LIST;
NOTE: TEST.PARENT.DATA is locked for exclusive access by you.
LOCK  test.parent CLEAR;
NOTE: TEST.PARENT.DATA is no longer locked by you.
%put &SYSLCKRC;
0
LOCK test.parent LIST;
NOTE: TEST.PARENT.DATA is not locked or in use by you or any other users.
```

In a multi-user environment, however, the LOCK LIST statement produces inaccurate output. When a lock—either shared or exclusive—is established on test.parent by the *first* SAS session and the code is rerun by a *second* SAS session, an invalid log is produced stating that test.parent is "not locked or in use by you or any other users." When the lock subsequently is attempted by the second session, however, an error results because the data set is in fact locked. The positive &SYSLCKRC return code 70031 accurately demonstrates that the lock could not be obtained, but only after spurious LIST results have been generated by SAS. This error was documented over ten years ago in Base SAS V8 but has not yet been remedied.[viii] The following log demonstrates the success of the first SAS session to secure a lock on the data set test.parent:

```
LOCK test.parent LIST;
NOTE: TEST.PARENT.DATA is not locked or in use by you or any other users.
LOCK test.parent;
NOTE: TEST.PARENT.DATA is now locked for exclusive access by you.
%put &SYSLCKRC;
0
LOCK test.parent LIST;
NOTE: TEST.PARENT.DATA is locked for exclusive access by you.
```

The following log demonstrates the unsuccessful attempt by a second session of SAS to lock the data set test.parent; note that the LOCK LIST statement twice incorrectly states that the data set is not locked, when in fact it remains locked by the first SAS session:

```
LOCK test.parent LIST;
NOTE: TEST.PARENT.DATA is not locked or in use by you or any other users.
LOCK test.parent;
ERROR: A lock is not available for test.parent.
%put &syslckrc;
70031
LOCK test.parent LIST;
NOTE: TEST.PARENT.DATA is not locked or in use by you or any other users.
```

Despite this error, the LOCK LIST statement fortunately has no functional responsibility in Base SAS, and only exists to echo lock status to the user via the log. Users are cautioned, however, not to rely on LOCK LIST to determine file lock status because it fails to recognize SAS data sets that are locked through external sessions of SAS. For this reason, issuance of the LOCK statement and subsequent testing of the &SYSLCKRC macro variable is the only way to assess lock status through the LOCK statement. As discussed below, lock status also can be assessed reliably through the OPEN and FOPEN functions, which can be implemented either through a DATA step or by invoking them through the %SYSFUNC macro command.

TESTING AND IMPLEMENTATION OF IMPLICIT LOCKS THROUGH THE DATA STEP

Whereas the LOCK statement relies on an explicit request to invoke an exclusive lock, DATA steps and procedures implicitly test locks and invoke either a shared or exclusive lock. The following code is used in four examples to demonstrate the interaction between two simultaneous sessions of SAS that are competing for concurrent access to the same data two sets.

```
DATA test.child;            /*requires an exclusive lock to create*/
    SET test.parent;        /*requires a shared lock to read*/
RUN;
```

If the data set test.parent already has an exclusive lock when the code is run, test.parent cannot be read by the second session and thus test.child cannot be created. The following log and error are produced:

```
DATA test.child;
    SET test.parent;
ERROR: A lock is not available for TEST.PARENT.DATA.
RUN;
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set TEST.CHILD may be incomplete. When this step was
    stopped there were 0 observations and 0 variables.
WARNING: Data set TEST.CHILD was not replaced because this step was
stopped.
```

If the data set test.child already has an exclusive lock when the code is run, test.child cannot be modified (i.e., overwritten) so test.child cannot be created. Thus, if an exclusive lock is maintained on a data set, it can neither be read nor modified by a concurrent second process. The following log and error are produced:

```
DATA test.child;
    SET test.parent;
RUN;
ERROR: A lock is not available for TEST.CHILD.DATA
NOTE: The SAS System stopped processing this step because of errors.
```

If the data set test.child already has a shared lock when the code is run, test.child cannot be modified (i.e., overwritten) so test.child cannot be created. The error differs from the above example in which the lock was exclusive; however, the result is the same and the data set cannot be created. The following log and error are produced:

```
DATA test.child;
    SET test.parent;
RUN;
ERROR: A lock is not available for TEST.CHILD.DATA.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set TEST.CHILD was only partially opened and will not be
saved.
```

In the final example, if the data set test.parent already has a shared lock when the code is run, the code executes correctly. This demonstrates that multiple processes concurrently can lock the same data set so long as all locks are shared. In the two sessions, neither user is even made aware that a separate process is also accessing the data set test.parent.

```
DATA test.child;
    SET test.parent;
RUN;
NOTE: There were 10 observations read from the data set TEST.PARENT.
NOTE: The data set TEST.CHILD has 10 observations and 22 variables.
```

Although the above four examples solely described the DATA step, identical errors would be produced by the following SORT procedure, were it applied to the same four scenarios:

```
PROC SORT DATA=test.parent        /*requires a shared lock to read*/
    OUT=test.child;               /*requires an exclusive lock to create*/
```

```
BY var;
RUN;
```

I/O FUNCTIONS OPEN AND FOPEN

The I/O functions OPEN and FOPEN, while also eliciting implicit locks, differ from DATA steps and procedures because functions generate return codes in lieu of errors when a lock cannot be achieved. Return codes allow these functions to be implemented easily into exception handling routines that dynamically reroute process flow and facilitate process success.

The function OPEN is utilized to gain read-only access to a data set, which implicitly produces a shared lock that lasts through the duration of the DATA step and is released thereafter. If successful, the return code is set equal to “1”, with subsequent successful OPEN functions within the same DATA step producing return codes incremented by one. If unsuccessful, the return code will equal “0”. The accompanying function CLOSE can be utilized after OPEN to ensure the data set is immediately available for further lock testing or processing, but otherwise is not required since the data set will be closed automatically when the data step terminates. Note that in previous versions of SAS, the OPEN function was also able to generate read-write access as well and thus could produce an exclusive lock. Past solutions to the file collision conundrum fail today because of this modification to the OPEN function.

In contrast to OPEN, the FOPEN function can be utilized to gain either read-only or read-write access to a file that can be but is not necessarily a SAS data set. The UPDATE option forces the read-write access level and thus can be utilized to generate and test for exclusive locks on data sets. If successful, the return code is set equal to “1”, with subsequent successful FOPEN functions within the same DATA step producing return codes incremented by one. If unsuccessful, the return code will equal “0”. Because FOPEN can operate with non-SAS files, the additional step of declaring the referenced file through the FILENAME statement is required. The accompanying function FCLOSE can be utilized after FOPEN to ensure the data set immediately is available for further lock testing or processing but, like CLOSE, is not required.

The following examples illustrate the behavior and interaction of OPEN and FOPEN statements in a single-session SAS environment. The data set test.parent has no locks when the code is invoked. In the first example, a request is first made for an exclusive lock of the data set test.parent, followed by a request for a shared lock. The exclusive lock is successful, but the subsequent shared lock cannot be obtained because the data set is still locked exclusively by the FOPEN function. This illustrates the critical point that although a session may have a data set locked, subsequent I/O functions that attempt a duplicate lock will fail.

```
%LET lib=test;
%LET tab=parent;
%LET loc=%SYSFUNC(PATHNAME(&lib))\&tab..sas7bdat;
FILENAME myfile "&loc";
DATA _null_;
    excl=FOPEN('myfile','U');     /* succeeds */
    shared=OPEN("&lib..&tab");    /* fails because FOPEN maintains
          the lock */
    PUT "EXCLUSIVE: " excl "  SHARED: " shared;
RUN;
EXCLUSIVE: 1   SHARED: 0
```

If the order of the FOPEN and OPEN functions are swapped, the results also are reversed; OPEN is successful and FOPEN is unsuccessful. As with the DATA step examples above, this illustrates that an exclusive lock cannot be obtained if a shared lock already exists for that data set. In the DATA step examples above, a multi-user environment was required to produce this type of error, whereas the functions OPEN and FOPEN mutually can preclude success of each other in a single-session environment.

```
shared=OPEN("&lib..&tab");     /* succeeds */
excl=FOPEN('myfile','U');      /* fails because OPEN maintains the lock */
EXCLUSIVE: 0   SHARED: 1
```

If instead two FOPEN functions are attempted on the same data set, the first will succeed and the second will fail.

```
excl1=FOPEN('myfile','U');
excl2=FOPEN('myfile','U');
PUT "EXCLUSIVE 1: " excl1 "  EXCLUSIVE 2: " excl2;
EXCLUSIVE 1: 1   EXCLUSIVE 2: 0
```

If the LOCK statement is invoked to exclusively lock a data set, potentially surprising results emerge. One might think that if an exclusive lock were desired, a pre-emptive LOCK statement would establish that lock before FOPEN was called; this is false. The LOCK statement does gain an exclusive lock on the data set; however, when FOPEN subsequently is invoked, it is unable to re-lock and open a file stream and thus fails. The OPEN function, however, can gain a shared lock despite the explicit lock from the LOCK statement, and thus is successful. The following results are produced:

```
LOCK test.parent
%LET lib=test;
%LET tab=parent;
%LET loc=%SYSFUNC(PATHNAME(&lib))\&tab..sas7bdat;
FILENAME myfile "&loc";
DATA _null_;
    excl=FOPEN('myfile','U');
    shared=OPEN("&lib..&tab");
    PUT "EXCLUSIVE: " excl "  SHARED: " shared;
RUN;
EXCLUSIVE: 0   SHARED: 1
```

While the above I/O function examples have described a single-session environment, the final example illustrates performance in a multi-user environment. Given two sessions of SAS in which the LOCK statement exclusively locks the data set test.parent in the first session, the following code is submitted in the second session, demonstrating that neither FOPEN nor OPEN can gain a lock if an exclusive lock already is held through a separate session of SAS:

```
%LET lib=test;
%LET tab=parent;
%LET loc=%SYSFUNC(PATHNAME(&lib))\&tab..sas7bdat;
FILENAME myfile "&loc";
DATA _null_;
    excl=FOPEN('myfile','U');
    shared=OPEN("&lib..&tab");
    PUT "EXCLUSIVE: " excl "  SHARED: " shared;
RUN;
EXCLUSIVE: 0   SHARED: 0
```

THE SAVIOR: %SYSFUNC(OPEN)

The original (unpublished) version of %LOCKITDOWN operated by testing exclusive locks with the FOPEN function, and shared locks with the OPEN function. For example, if a shared lock were desired on the data set test.locked, the FOPEN function repeatedly would attempt to open the file through a _null_ DATA step. Once FOPEN generated a return code greater than zero, this indicated the file no longer was exclusively used by other sessions of SAS, thus the spinlock would exit, %LOCKITDOWN would terminate, and control would return to process the DATA step or procedure that required the shared lock. The flow chart for the original (incorrect) version of %LOCKITDOWN is presented in Figure 1.

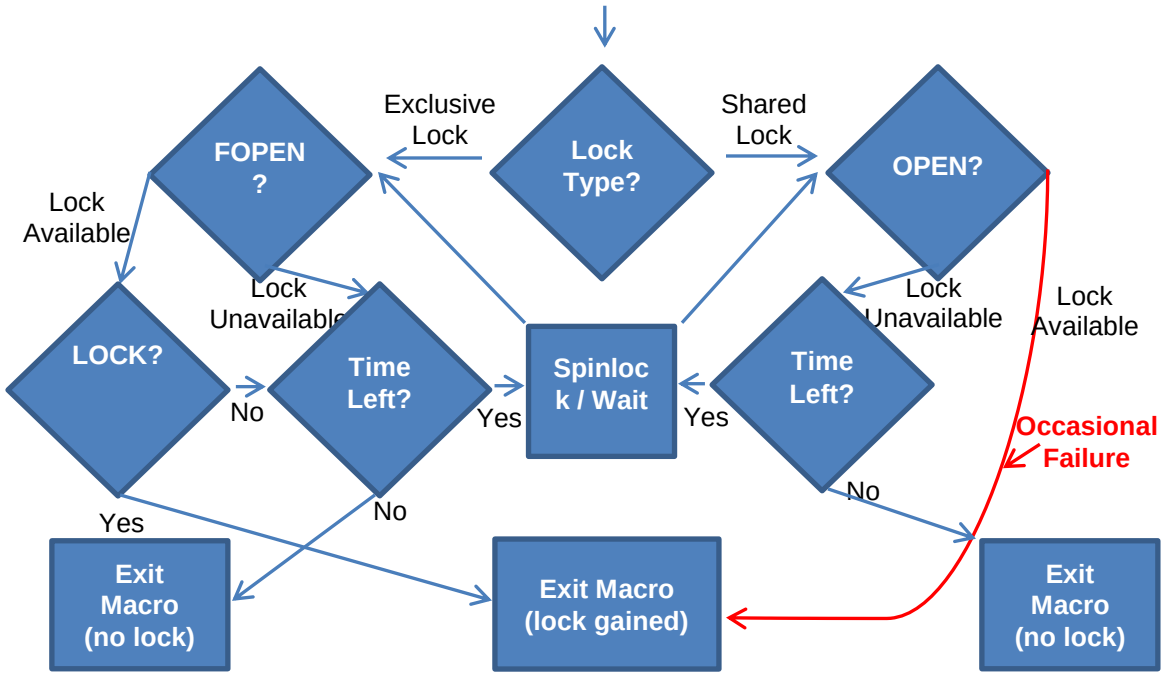

Figure 1. Original (Incorrect) %LOCKITDOWN Process Flow

The original macro functioned in most circumstances; however, in stress testing under duress—when multiple processes simultaneously attempt to access the same data set in rapid succession—failure occurred every few thousand iterations. The OPEN function accurately was waiting until a shared lock was achievable but, because OPEN was called from within the DATA step, the shared lock that OPEN achieved was released when the DATA step concluded within the macro. Thus, in the micro-seconds between lock release and macro termination, a concurrent session could sneak in and achieve an exclusive lock on the same data set, leading to failure of the first session.

The fault was remedied through implementation of the OPEN function within the %SYSFUNC macro function. Because %SYSFUNC allows OPEN to be called from outside a DATA step, the shared lock that %SYSFUNC achieves persists after macro termination, thus maintaining the lock until the subsequent DATA step or procedure gains an implicit shared lock. With the micro-second gap eliminated, so too was the potential for failure. The downside of this modification is that %SYSFUNC(OPEN) requires a corresponding %SYSFUNC(CLOSE) function once the data set lock is no longer needed. Because the data set ID must be referenced in the CLOSE statement, the macro %LOCKCLR (see Appendix A) is provided as a simple method to close up to ten shared data sets that have been opened.

On equally rare occasions, a similar gap also occurs after achieving an exclusive lock through the FOPEN function but, because the status of the LOCK statement is assessed through the &SYSLCRC macro variable, execution inside the macro returns to the spinlock, rather than exiting and failing. For this reason, the SAS log occasionally will demonstrate the failure of the LOCK statement. However, controlled within the confines of the %LOCKITDOWN macro, appropriate exception handling ensures the spinlock is re-engaged and no process failure results. Notwithstanding, the appearance of these lock "errors" in the log may frustrate SAS users as well as automated log checking routines, but they should in fact be treated as "notes" instead as no enduring error in the process flow has actually occurred. The updated %LOCKITDOWN macro is demonstrated in Figure 2, and differs only because OPEN is called within the %SYSFUNC macro function.

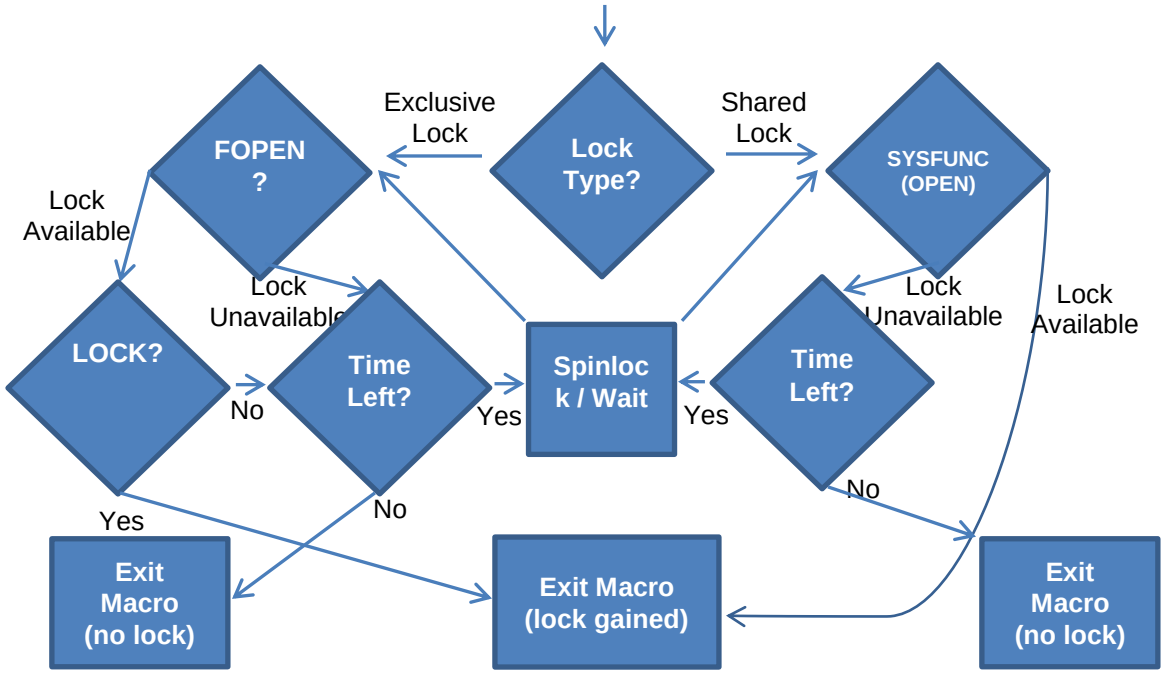

Figure 2. Correct %LOCKITDOWN Process Flow

THE %LOCKITDOWN MACRO SOLUTION

John Leveille's and Jimmy Hunnings' 2005 seminal paper "Play Nice with Others: Waiting for a Lock on SAS Table"[ix] contains the underlying spinlock and busy-waiting logic utilized in %LOCKITDOWN. In a spinlock, data availability repeatedly is tested until a lock is achieved or the process ends due to process timeout. After each unsuccessful attempt, the SLEEP function—or CALL SLEEP function in UNIX environments—causes SAS to wait a parameterized amount of time before a subsequent attempt and, if the maximum amount of time has elapsed, the spinlock exits. Due to modifications already discussed in current SAS functionality of the OPEN function, their solution unfortunately no longer is viable, although its logic forms the underpinnings of the %LOCKITDOWN macro.

In traditional SAS solutions for data availability testing, the LOCK statement is utilized both to test and eventually achieve an exclusive lock before data processing. While effective, this unnecessarily produces exclusive locks on data sets for which a shared lock will suffice, thus preventing other processes from obtaining shared locks on the same data set. Consider the production data set prod.thebigtamale that is utilized extensively by an organization for data analytics. A single, serialized process may create the data set once each night, but throughout the day, hundreds of concurrent processes may read this data set, accessing it through shared locks. The LOCK statement is required to gain an exclusive lock each night during file creation, but should not be utilized during the day for read-only access because this would prevent additional processes from reading the data concurrently. Thus, overuse of the LOCK statement can create a tremendous inefficiency, as processes must wait in line for unnecessary exclusive locks to be released.

The code for the %LOCKITDOWN macro is included in Appendix A, and the macro is invoked with the following code:

```
%macro LOCKITDOWN(lockfile= /* data set in LIBRARY.DATASET or DATASET
        format */,
    sec=5 /* interval in seconds after which data set is re-tested*/,
    max=300 /* maximum seconds waited until timeout */,
    type=W /* either (R) or (W) for READ-only or read-WRITE lock */,
    canbemissing=N /* either (Y) or (N), indicating if data set can not
        exist */);
```

The parameters for %LOCKITDOWN include:

- LOCKFILE – Data sets typically only need to be locked when they reside in permanent SAS libraries that are accessible to other users or to concurrent SAS sessions. Notwithstanding, %LOCKITDOWN is flexible enough to test and lock data sets in the WORK library, by simply omitting the library token in the LOCKFILE parameter.
- SEC – If a lock cannot be achieved, the SAS session will sleep a number of seconds before reattempting. Thus, this parameter also acts to prioritize sessions, some of which simultaneously may be waiting for the same locked data set. For example, a session with a 1 second sleep interval will be approximately ten times more likely to achieve a lock before a simultaneous session attempting access but using a 10 second sleep interval. In this manner, the most critical processes can be prioritized over less significant ones by reducing the SEC parameter.
- MAX – After an unacceptable maximum delay, the *event*—a locked file—becomes an *exception*. When transactional or standardized data sets are being processed, process time estimation typically is highly correlated with file size and thus can be estimated with some degree of certainty. If multiple processes are anticipated to be attempting access to the same data set, however, the MAX parameter may need to be increased to account for a process that may have to wait for several other processes first to obtain and then release locks on the same data set.
- TYPE – Discussed above, the appropriate lock type must be chosen contextually. Failure to select the correct lock type could lead to a data set obtaining only read-only access and producing a runtime error if the subsequent process actually requires read-write access.
- CANBEMISSING – This indicates whether a data set must exist. For example, often in dynamic processes, a data set may be created during the first iteration, and subsequently modified repeatedly. This informs %LOCKITDOWN to first determine file existence and, if no data set exists, to proceed without obtaining a lock (since a nonexistent file cannot be locked!)

%LOCKITDOWN IMPLEMENTATION AND EXCEPTION HANDLING

In the previous example, the data set prod.thebigtamale is created nightly and is the primary data product for an organization. Analysts subsequently rely on the data throughout the day for analysis through repeated read-only access. The following simplified code creates the data set each night:

```
DATA prod.thebigtamale;
   MERGE blah moreblah;
   BY ID;
RUN;
```

But what happens when an analyst works late one night and is running a process that accesses thebigtamale (through an implicit read-only process) while the above process attempts subsequently to re-create the data set? The hapless analyst continues undaunted in his work, leaving at 12:30am to head to—you guessed it—In N-Out Burger® for well-deserved indulgence. He returns, however, in the morning to learn that while *his* code succeeded, it caused the above nightly update to fail, producing cascading effects throughout his organization, in addition to the following error log:

```
ERROR: A lock is not available for PROD.THEBIGTAMALE.DATA.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set PROD.THEBIGTAMALE was only partially opened and will
not be saved.
```

The analyst, however, is not really to blame. Rather, the persistent process that creates thebigtamale data set was not robust enough because it failed to anticipate the event of the data set being locked at midnight. Yes, the operations team had minimized the likelihood of this failure by running the process overnight, but their solution was not robust enough. The below invocation of %LOCKITDOWN will determine the availability of the data set before proceeding and, if thebigtamale has been deleted (intentionally or accidentally), the code dynamically assesses this and continues. Note that because the LOCK statement invokes an explicit lock, this lock also must be removed explicitly using the &LOCKCLR macro variable (which calls the respective LOCK CLEAR statement.)

```
%LOCKITDOWN (lockfile=prod.thebigtamale, sec=1, max=600, type=W,
    canbemissing=Y);
DATA prod.thebigtamale;
    SET blah;
RUN;
&lockclr;    */ the macro VARIABLE is used for exclusive lock clearing */
```

If the data set initially is unlocked, the data set will be produced without error. No change in process occurred from the above code, yet the program was executed with the additional assurance of having first obtained an explicit lock.

```
NOTE: PROD.THEBIGTAMALE.DATA is now locked for exclusive access by you.
DATA prod.thebigtamale;
    SET blah;
RUN;
NOTE: There were 200000 observations read from the data set WORK.BLAH.
NOTE: The data set PROD.THEBIGTAMALE has 200000 observations and 100
variables.
&lockclr;
NOTE: PROD.THEBIGTAMALE.DATA is no longer locked by you.
```

The overnight production code is now more robust and reliable with because it accounts for the potential event that some diligent analyst is viewing the data set at midnight and has locked the file. In this case, after repeated attempts to gain an exclusive file lock for ten minutes, %LOCKITDOWN terminates and produces the following output log:

```
LOCKOUT failed after 600.09 seconds to gain access to prod.thebigtamale
```

The code above will *catch* the exception and thus prevent a LOCK error from stopping the process; it even notifies the operations team via the log that a lock could not be obtained. It doesn't, however, *handle* the exception. Instead, any code that followed the DATA step still would have executed and, because thebigtamale could not be created, that subsequent code probably would have failed or produced invalid results. Thus, proper exception handling is contextual and must follow business rules that prescribe the sequence of processes to follow when a specific exception occurs.

The following revised code finally demonstrates contextual exception handling, in that failure to lock the data set both is caught and handled. In this example, if %LOCKITDOWN generates any error messages, the macro %DOSTUFF will terminate due to the %RETURN function, preventing subsequent, derivative process failure:

```
%macro dostuff;
    %LOCKITDOWN(lockfile=prod.thebigtamale, sec=1, max=600, type=W,
          canbemissing=Y);
    DATA prod.thebigtamale;
          SET blah;
    RUN;
    &lockclr;
    %if %sysevalf(%length(&lockerr)>1) %then %return;
%mend;
```

DEPLOYING LOCKS TO MULTIPLE FILES

The above examples each referenced only a single data set within a permanent library, thus requiring only one %LOCKITDOWN implementation per process. However, in serialized extract transform load (ETL) data flows, complex processes may produce several incremental or disparate data sets that are saved to permanent libraries, thus requiring successive calls to %LOCKITDOWN to build reliable infrastructure. Consider the below process that incorporates no exception handling:

```
DATA prod.thebigtamale;
    SET prod.thebigenchilada;
RUN;
```

In this example, the DATA step will fail if either thebigtamale already has an exclusive lock or thebigenchilada already has an exclusive lock. Thus, in order to execute successfully, thebigtamale must obtain an exclusive lock and thebigenchilada must obtain a shared lock. To make the code more robust, the following dual implementation of %LOCKITDOWN can be used:

```
%macro tamale_makin;
%LOCKITDOWN(lockfile=prod.thebigtamale, sec=1, max=600, type=W,
    canbemissing=Y);
%if %eval(%length(&lockerr)>0) %then %goto err;
%else %do;
    %LOCKITDOWN(lockfile=prod.thebigenchilada, sec=1, max=600, type=R
                canbemissing=N);
    %if %eval(%length(&lockerr)>0) %then %do;
          lock prod.thebigtamale clear; *required because it is type W;
          %goto err;
          %end;
    %else %do;
          DATA prod.thebigtamale;
                SET prod.thebigenchilada;
          RUN;
          %lockclr; */ macro FUNCTION is used for shared lock clearing */
          LOCK prod.thebigtamale CLEAR;
          %end;
    %end;
%err:;
%mend;
```

The above revised code now represents a far more robust, reliable solution. It plays well with others, waits patiently for its turn for both data sets and, if it doesn't get what it wants, it gracefully terminates rather than continuing to run and causing a commotion and subsequent failures. A further improvement, especially for cases in which several data sets need to be locked, would be to develop a dynamic round-robin approach that tests for file availability and, if not immediately achieved, progresses immediately to test the next data set. Although outside the scope of this text, this approach would gain substantial efficiency over the above serialized approach if several data sets need to be locked at once.

PROC DATASETS AND LOCKED DATA SETS

One final idiosyncrasy of the LOCK statement should be mentioned which occurs when the DATASETS procedure is applied to data sets that are locked. Consider the following code that explicitly locks thebigtamale data set:

```
LOCK prod.thebigtamale;
```

In a multi-user environment, if a second session of SAS now subsequently runs the DATASETS procedure, interesting results are achieved. Because DATASETS iterates through all CONTENTS and only stops after the QUIT statement, it actually can produce both an error and seemingly valid yet incomplete results. Implementation of the ERRORABEND system option will cause SAS to terminate after encountering the first error, and can be used to avoid the potential confusion that the DATASETS procedure can produce when data sets are locked. The following procedure is run to generate an output data set work.libout that contains records of each data set contained in the library prod:

```
PROC DATASETS LIBRARY=prod;
    CONTENTS DATA=_all_ OUT=work.libout NOPRINT;
    QUIT;
RUN;
```

Because thebigtamale was already locked when DATASETS was executed, its metadata will not be captured in the resultant file work.libout, and the following error is produced in the log:

```
ERROR: A lock is not available for WORK.THEBIGTAMALE.DATA.
NOTE: The data set WORK.LIBOUT has 1120 observations and 40 variables.
NOTE: Statements not processes because of errors noted above.
```

```
NOTE: The SAS System stopped processing this step because of errors.
```

The DATASETS procedure actually does create the requested output data set, but omits reference to any data sets that were locked at the time it was executed. The SAS automatic macro variable &SYSERR also will only record the last error that occurs in the procedure. For the majority of procedures this is effective, but for PROC DATASETS, this may lead a user (or process) that ignores the log and, in viewing only the &SYSERR value, incorrectly infers that only one data set was locked when in fact numerous data sets could have been locked. PROC SQL as well can be used to access the SAS dictionary through the SASHELP.VTABLE, but this too fails to recognize SAS data sets that are locked. Thus, in producing code that dynamically generates a list of all data sets contained within a SAS library, it may be necessary to employ external methods (such as a DOS pipe command) to ensure accuracy and completeness of the resultant data set list.

SURPASSING SERIALIZATION TO CONQUER CURRENCY

The utility of the %LOCKITDOWN macro is unquestionable as it overcomes a primary obstacle of file collisions resultant from failed concurrent access attempts. More than ensuring stable data processes, however, this workhorse can facilitate efficiency through advanced process design. Once a serialized process has reached a reliable state and no longer is failing due to locked data sets and other unhandled events or exceptions, a common subsequent request or priority is to make the process go faster. By modularizing components and running them in parallel through concurrent SAS sessions, %LOCKITDOWN can enable “simulated simultaneous use” of data sets, as processes access and update shared control tables in micro-seconds, allowing transfer of performance metrics and data-driven directives between concurrent sessions of SAS.

Imagine a serialized process flow that iterates through a list of six data sets and sorts them through dynamic macro processing. The *coding* may have been efficient because it utilized dynamic macros to reduce redundancy and improve readability, but the *performance* is terribly inefficient because all processing is serialized in a phase-gate approach. Now imagine a subtle modification in which a single control table directs the selection of data sets to be sorted through data-driven processing. After each data set sort is completed, the control table is locked for a split second while the sort completion is recorded and the next data set to be sorted is selected. Rather than having a single, serialized process running, multiple instances of SAS can now execute the identical code which completes in a fraction of the time. Figure 3 demonstrates a typical serialized process flow that sorts six data sets, with relative process time represented by arrow length.

Figure 3. Serialized Process Flow Executed in Single-Session Environment

By creating dynamic, intelligent code that can be spawned simultaneously through multiple SAS instances, parallel processing can be achieved. Given the same data sets from the above example, the same sort functions executed by three instances of identical code running in parallel will complete in less than half the time because the control table is directing traffic. Figure 4 demonstrates the implementation of the identical sort but executed in a parallel processing environment.

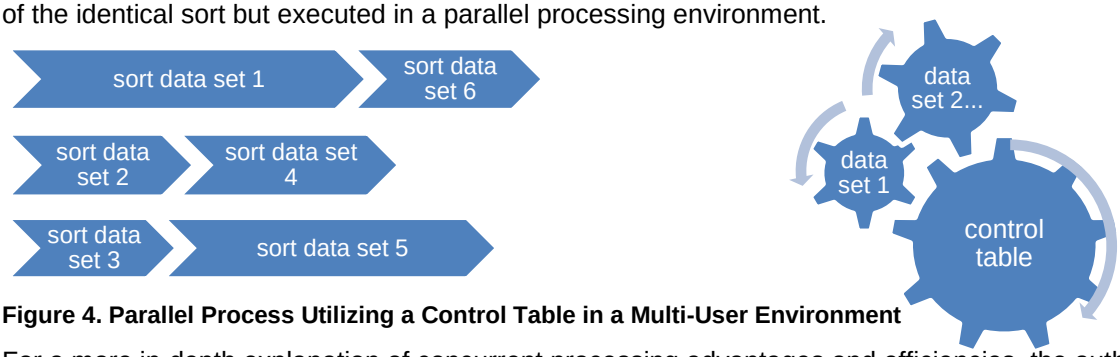

Figure 4. Parallel Process Utilizing a Control Table in a Multi-User Environment

For a more in-depth explanation of concurrent processing advantages and efficiencies, the author has demonstrated the implementation of %LOCKITDOWN to facilitate parallel processing in an ETL

infrastructure[x]. The author also has demonstrated a quality control dashboard that monitors data availability, structure, and content in near-real-time[xi], with both implementations relying on fuzzy logic control tables that direct data-driven processing and which rely on the %LOCKITDOWN macro.

CONCLUSION

Robust, reliable data processes require the assurance that execution will succeed or, when failure is unavoidable, that notice of the failure is communicated to users, administrators, and other stakeholders. When exceptional events do occur, these must be both recognized and handled to ensure graceful program termination and to prevent further process failure or invalid results. As an infrastructure matures and becomes increasingly more complex, the likelihood of data access collisions—one of the most common failures in Base SAS processes—also will increase. These failures, however, can be avoided with the implementation of the %LOCKITDOWN macro. And, while data sets occasionally will still be unavailable because they remain locked for an unacceptable period of time, through dynamic exception handling routines, the %LOCKITDOWN macro maximizes process reliability and ultimately data quality. Moreover, implementation of %LOCKITDOWN can facilitate the creation of dynamic, concurrent processing models that substantially improve process performance and efficiency.

REFERENCES

[i] Hughes, Troy Martin. 2014. Reliably Robust: Best Practices for Quality Assurance and Quality Control in SAS Software Design. Southeast SAS Users Group (SESUG).

[ii] SAS, Inc. Products and Solutions: SAS/SHARE. Retrieved from: http://www.sas.com/products/share/.

[iii] Hughes, Troy Martin. 2014. Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS. Midwest SAS Users Group (MWSUG).

[iv] SAS, Inc. SAS/SHARE 9.3 User's Guide. Locking SAS Data Objects. Retrieved from: http://support.sas.com/documentation/cdl/en/shrref/63064/HTML/default/viewer.htm#n0o0x58g1mpnfkn11jcxxqdfjdxb.htm

[v] SAS, Inc. Locking and SAS Data Object Hierarchy. SAS/SHARE 9.3 User's Guide. Retrieved from http://support.sas.com/documentation/cdl/en/shrref/63064/HTML/default/viewer.htm#n0pwd6wb19m3ehn1juwjd8nwtrp9.htm

[vi] SAS, Inc. Defaults for Selected SAS Operations. SAS/SHARE 9.3 User's Guide. Retrieved from http://support.sas.com/documentation/cdl/en/shrref/63064/HTML/default/viewer.htm#p0iqymuxvfhhnjn1u7msnsiej9mm.htm

[vii] SAS, Inc. LOCK Statement. SAS 9.3 Statements: Reference. Retrieved from http://support.sas.com/documentation/cdl/en/lestmtsref/63323/HTML/default/viewer.htm#p0pc4mpj7xzxs4n1257fgz2tkuyb.htm

[viii] SAS, Inc. Problem Note 2859: LOCK statement or LOCK function with LIST or QUERY options may report locks incorrectly. SAS Knowledge Base / Samples and SAS Notes. Retrieved from: http://support.sas.com/kb/2/859.html.

[ix] Leveille, John and Hunnings, Jimmy. 2005. Play Nice with Others: Waiting for a Lock on SAS Table. The Pharmaceutical Industry SAS Users Group.

[x] Hughes, Troy Martin. 2014. Calling for Backup When Your One-Alarm Becomes a Two-Alarm Fire: Developing Base SAS Data-Driven, Concurrent Processing Models through Fuzzy Control Tables that Maximize Throughput and Efficiency. South Central SAS Users Group (SCSUG).

[xi] Hughes, Troy Martin. 2014. Will You Smell Smoke When Your Data Are on Fire? The SAS Smoke Detector: Installing a Scalable Quality Control Dashboard for Transactional and Persistent Data. Midwest SAS Users Group (MWSUG).

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com

APPENDIX A. %LOCKITDOWN MACRO

```
%macro LOCKITDOWN(lockfile= /* data set in LIBRARY.DATASET or DATASET
    format */,
    sec=5 /* interval in seconds after which data set is re-tested */,
    max=300 /* maximum seconds waited until timeout */,
    type=W /* either (R) or (W) for READ-only or read-WRITE lock */,
    canbemissing=N /* either (Y) or (N), indicating if data set can not
          exist */);
%local i;
%local lib;
%local tab;
%local starttime;
%local max;
%local loc;
%global dsid;
%global lockerr;
%global lockclr;
%let lockerr=;
%let lockclr=;
%let dsid=;
%let type=%upcase(&type);
%if &type=READ %then %let type=R;
%else %if &type=WRITE %then %let type=W;
%let canbemissing=%upcase(&canbemissing);
%if &canbemissing=YES %then %let canbemissing=Y;
%else %if &canbemissing=NO %then %let canbemissing=N;
%if &SYSSCP=WIN %then %let sys=WIN;
%else %let sys=UNIX;
%let i=1;
* determine whether a libname is included in the filename parameter;
%do %while(%length(%scan(&lockfile,&i,.))>0);
    %let i=%eval(&i+1);
    %end;
%if &i=2 %then %do;
    %let lib=work;
    %let tab=%scan(&lockfile,1,.);
    %end;
%else %if &i=3 %then %do;
    %let lib=%scan(&lockfile,1,.);
    %let tab=%scan(&lockfile,2,.);
    %end;
%else %do;
    %let lockerr=LOCKITDOWN failed because too many levels in file name;
    %goto err;
    %end;
* determine whether libname, data set name, and type are valid and present;
%if %sysfunc(libref(&lib))^=0 %then %do;
    %let lockerr=LOCKITDOWN failed because library %upcase(&lib) not
assigned;
    %goto err;
    %end;

%if %sysfunc(exist(&lib..&tab))^=1 %then %do;
    %if &canbemissing=N %then %do;
```

```
          %let lockerr=LOCKITDOWN failed because data set
%upcase(&lib..&tab)
                       does not exist;
          %goto err;
          %end;
    %else %if &canbemissing=Y %then %do;
          %goto noerr;
          %end;
    %end;
%if &type^=R and &type^=W %then %do;
    %let lockerr=LOCKITDOWN failed because value for TYPE must be either R
or W;
    %goto err;
    %end;
%if &canbemissing^=Y and &canbemissing^=N %then %do;
    %let lockerr=LOCKITDOWN failed because CANBEMISSING must be Y or N;
    %goto err;
    %end;
* lock testing;
%let starttime=%sysfunc(datetime());
%let loc=%sysfunc(pathname(&lib))\&tab..sas7bdat;
filename myfile "&loc";
%do %until(%eval(&dsid>0) or
%sysevalf(%sysfunc(datetime())>&starttime+&max));
    %if &type=W %then %do;
          data _null_;
                dsid=fopen('myfile','u');
                call symput('dsid',dsid);
          run;
          %if %eval(&dsid>0) %then %do;
                lock &lib..&tab;
                %if %eval(&syslckrc^=0) %then %do;
                      %put LOCK FAILED;
                      %let dsid=0;
                      %end;
                %end;
          %end;
    %else %if &type=R %then %let dsid=%sysfunc(open(&lib..&tab));
    %if %eval(&dsid^=0) %then %do;
          %if &type=W %then %do;
                %let lockclr=lock &lib..&tab clear;;
                %end;
          %end;
    %else %do;
          %let dsid=0;
          %put SLEEPING &sys;
          %if &sys=WIN %then %let sleeping=%sysfunc(sleep(&sec));
          %else %if &sys=UNIX %then %do;
                data _null_;
                      call sleep(&sec,1);
                run;
                %end;
          %end;
    %end;
%if &dsid=0 %then %do;
    %let lockerr=LOCKITDOWN failed after %sysevalf(%sysfunc(datetime())-
```

```
          &starttime) seconds to gain %sysfunc(ifc(&type=W,an exclusive,a
          shared)) lock on &lockfile;
    %let lockclr=;
    %end;
%err: %put &lockerr;
%noerr:;
%mend;

%macro lockclr;
    %local i;
    %do i=1 %to 10;
          %let x=%sysfunc(close(&i));
          %end;
%mend;
```