Paper 1465-2017

Stress Testing and Supplanting the SAS® LOCK Statement: Implementing Mutex Semaphores To Provide Reliable File Locking in Multiuser Environments To Enable and Synchronize Parallel Processing

Troy Martin Hughes

ABSTRACT

The SAS® LOCK Statement was introduced in SAS 7 with great pomp and circumstance, as it enabled SAS software to lock data sets exclusively. In a multiuser or networked environment, an exclusive file lock prevents other users and processes from accessing and accidentally corrupting a data set while it is in use. Moreover, because file lock status can be tested programmatically with the LOCK statement return code (&SYSLCKRC), data set accessibility can be validated before attempted access, thus preventing file access collisions and facilitating more reliable, robust software. Notwithstanding the intent of the LOCK statement, stress testing demonstrated in this text illustrates vulnerabilities in the LOCK statement that render its use inadvisable due to its inability to lock data sets reliably outside of the SAS/SHARE environment. To overcome this limitation and enable reliable data set locking, a methodology is demonstrated that utilizes semaphores (i.e., flags) that indicate whether a data set is available or is in use, and mutually exclusive (mutex) semaphores that restrict data set access to a single process at one time. With Base SAS file locking capabilities now restored, this text further demonstrates control table locking to support process synchronization and parallel processing. The LOCKSAFE macro demonstrates a busy-waiting (or spinlock) design that tests data set availability repeatedly until file access is achieved or the process times out.

INTRODUCTION

The SAS LOCK statement was introduced in SAS 7 as a method to overcome file access collisions that can occur when two or more users or processes simultaneously attempt to achieve an exclusive lock on the same data set. Because exclusive locks are common and are required whenever a data set is created or modified, file locking functionality improved the robustness and reliability of SAS software by preventing runtime errors. For example, if a data set was programmatically assessed to be locked, the program could enter a busy-waiting cycle in which it repeatedly tested data set availability until the required lock—either shared (i.e., read-only) or exclusive (i.e., read-write)—was achieved or the process timed out.

One of the biggest advantages of the LOCK statement was its ability to facilitate parallel processing via a control table—that is, a SAS data set that dynamically drives program flow. For example, a program could exclusively lock a control table, modify its data, and subsequently unlock the control table so that a separate program running in a separate SAS session could access the data set and perform some data-driven task. With multiple SAS sessions able to communicate via data passed through shared control tables, divide-and-conquer and other parallel processing paradigms could be designed that substantially improve software execution speed. This communication is impossible, however, without shared control tables that can be reliably locked while they are modified.

With the introduction of SAS 9.4, undocumented in the *What's New in SAS® 9.4* publication was the fact that SAS had deprecated most of the LOCK statement functionality. This change is obvious when the *SAS® 9.3 Statements: Reference* and *SAS® 9.4 Statements: Reference, Fourth Edition* are compared. The promise that "no other SAS session can read or write to the file until the lock is released" was removed from the LOCK statement documentation, replaced with a never-before-seen caveat that "to ensure that an exclusive lock is guaranteed across multiple SAS sessions, you must use SAS/SHARE." The era of secure SAS file locking had quietly come to an end.

The LOCK statement doesn't *always* fail—it just fails under certain stressful conditions that are exacerbated when locking and unlocking functions occur (on the same data set) in quick succession from different SAS sessions. Thus, for some software and purposes, the LOCK statement may operate successfully throughout software lifespan while in other cases the LOCK statement can sporadically fail for seemingly unknown (and undocumented) reasons. Stress

testing within this text demonstrates various failure patterns of the LOCK statement as well as the importance of performing stress testing in general as a quality assurance component of the software development life cycle (SDLC).

The solution introduced utilizes mutex semaphores that demonstrate whether a data set is locked, thus preventing file access collisions that occur from attempted (exclusive) simultaneous data set access. The semaphore for a data set is programmatically assessed before attempted data set access, similar to testing the &SYSLCKRC return code after attempting to lock a data set with LOCK. If the data set is available, the semaphore is modified to indicate its impending use, after which the data set can be securely accessed. Thus, the *semaphore* is the flag that indicates data set availability and the *mutex* is the logic that enforces that file locks be mutually exclusive.

The LOCKSAFE macro is introduced and automates this quality control (i.e., concurrency control) technique within a busy-waiting cycle that enables a process to wait until a data set is available (i.e., unlocked) or to terminate if data set access cannot be achieved within a parameterized amount of time. Because reliable control table locking is a critical component of many parallel processing models, the use of mutex semaphores to support reliable file locking can improve the speed, reliability, and robustness of software. Moreover, because the LOCK statement was widely implemented in the past as an attempt to solve this concurrency challenge and deliver added performance, a replacement solution is required now that LOCK has been shown to be woefully inadequate. LOCKSAFE has been tested on and is portable to the SAS Display Manager, SAS Enterprise Guide, and SAS University Edition.

WHY IS FILE LOCKING NECESSARY?

In a multiuser or multi-session environment, file access collisions occur when one SAS session holds a shared or exclusive lock on a data set while a second SAS session concurrently attempts to gain an exclusive lock on the same data set, or when one SAS session holds an exclusive lock on a data set while a second SAS session concurrently attempts to gain a shared lock on the same data set. For a thorough review of and differentiation between SAS shared and exclusive locks, please consult the author's text: *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*.[i] Unfortunately, the solution described implements the now-deprecated LOCK statement, rendering it unreliable in some situations in which rapid file locking occurs between two or more SAS sessions.

To give one example of the role of file locking, the following DATA step is executed, which requires an exclusive lock on the Test data set while it is created:

```
data test;
    length char $12;
    do i=1 to 10;
       char='SAS is rad!';
       output;
       end;
run;
```

To simulate file access collisions that occur in software that lacks robustness, two SAS sessions should be opened (within the SAS Display Manager or SAS Enterprise Guide) so that two programs can be executed concurrently. Note that the PERM library must first be initialized within each session, which may require modification based on user permissions and environment. In the first SAS session, execute the following code, which iteratively creates a nine-observation data set 1,000 times:

```
libname perm 'c:\perm'; * change to appropriate location;

%macro test_collide(dsn=, cnt=, session=);
proc printto log="%sysfunc(pathname(perm))\log&session..log" new;
run;
%do i=1 %to &cnt;
    %put START: &i;
    data &dsn;
```

```
        length char $12;
        do i=1 to 9;
           char='SAS is rad!';
           output;
           end;
    run;
    %put STOP;
    %end;
proc printto;
run;
%mend;

%test_collide(dsn=perm.test, cnt=1000, session=1);
```

Immediately after starting the program, execute the following code in the second SAS session so that the programs are executing concurrently in both sessions (noting that the full text of the previous TEST_COLLIDE macro will need to be copied into the second session):

```
libname perm 'c:\perm'; * change to appropriate location;

*** insert test_collide macro definition here ***;

%test_collide(dsn=perm.test, cnt=1000, session=2);
```

The second session performs the same actions as the first but records the results in a separate log file to ensure the logs of each respective session are not overwritten. When the programs in both sessions have completed, the following code should be run in the first session to parse and analyze the log file from the first session:

```
%macro readlog1();
data perm.readlog (keep=obs log);
    length tab $200 obs 8 log $2000;
    infile "%sysfunc(pathname(perm))\log1.log" truncover;
    input tab $200.;
    if _n_=1 then do;
       obs=0;
       log='';
       end;
    if substr(tab,1,6)='START:' then do;
       obs=substr(tab,7,12);
       log='';
       end;
    else if substr(tab,1,4)='STOP' then output;
    else do;
       if not(find(tab,"real time")>0 or find(tab,"cpu time")>0) and
             length(tab)>1 then do;
          log=strip(log) || ' *** ' || tab;
          end;
       end;
    retain log obs;
run;

proc freq data=perm.readlog order=freq;
    tables log / nocum;
run;
```

```
%mend;

%readlog1;
```

Each session attempts to create the PERM.Test data set 1,000 times but, because data set creation requires an exclusive lock and only one session can gain an exclusive lock at one time, the two processes take turns succeeding and failing. Table 1 demonstrates the frequencies of various execution paths, including successful (shown in black) and failed (shown in red) DATA steps from the first session, iterated 100,000 times and recorded in the PERM.Readlog data set.

Log (Session One Only)	Frequency	Percent
*** NOTE: The data set PERM.TEST has 9 observations and 2 variables. *** NOTE: DATA statement used (Total process time):	80,664	80.66
*** ERROR: A lock is not available for PERM.TEST.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** NOTE: DATA statement used (Total process time):	8,428	8.43
*** ERROR: A lock is not available for PERM.TEST.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** WARNING: The data set PERM.TEST was only partially opened and will not be saved. *** NOTE: DATA statement used (Total process time):	7,883	7.88
*** NOTE: The data set PERM.TEST has 9 observations and 2 variables. *** ERROR: Rename of temporary member for PERM.TEST.DATA failed. *** File can be found in c:\perm. *** NOTE: DATA statement used (Total process time):	1,694	1.69
*** NOTE: The data set PERM.TEST has 9 observations and 2 variables. *** ERROR: A lock is not available for PERM.TEST.DATA. *** NOTE: DATA statement used (Total process time):	874	0.87
*** ERROR: User does not have appropriate authorization level for file PERM.TEST.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** NOTE: DATA statement used (Total process time):	457	0.46

Table 1. Sample Frequency of 100,000 Iterations of TEST_COLLIDE Macro

In this example, the DATA step most commonly succeeds—approximately 80.7 percent (N=80,664) of the time. However, five distinct failure paths (represented by unique runtime error messages) occur when the two sessions collide and the DATA step fails in either session one or two. These file collision errors represent the rationale for implementing the LOCK statement to facilitate robustness and reliability, as demonstrated in the following sections.

STRESS TESTING RACE CONDITIONS

The International Organization for Standardization (ISO) defines *stress testing* as "testing conducted to evaluate a system or component at or beyond the limits of its specified requirements."[ii] Stress testing often anecdotally refers to testing a process or program incrementally to elicit and determine its failure point—the point at which either functional or performance failure begins to occur. In the previous example, the TEST_COLLIDE macro can be considered a stress test because it attempts to create a data set dozens of times per second in two competing SAS sessions—stressful conditions that might not occur in actual software implementation. By executing each DATA step thousands of times, stress testing aims to discover the full array of failure paths, including those less commonly encountered. For example, had stress testing included only 100 iterations, the "User does not have appropriate authorization level" error might not have been discovered due to its low frequency of occurrence.

From a risk management perspective, the five failure paths identified in Table 1 demonstrate a single risk (i.e., process failure) caused by a single threat (i.e., concurrent execution). The vulnerability that the threat exploits is commonly known as a *race condition*, in which two or more programs, processes, or threads independently attempt to access data or an object for which mutual exclusion is required but not enforced. Essentially, although multiple

processes may be *racing* to gain access to the same object, only one process can exclusively hold the object at one time, causing other processes to fail or produce invalid results. In object-oriented programming (OOP) languages that implement multithreaded processing, race conditions often result from a race to a shared variable (or other object) by separate threads of the same program. Because Base SAS does not enable SAS practitioners to write multithreaded processes (outside of the limited multithreaded capabilities introduced in the DS2 language), SAS race conditions most commonly describe independent programs operating in separate SAS sessions that attempt to access the same object concurrently. For example, concurrent programs attempting to write to the same data set, output file, or log file describe race conditions that, when exploited, cause runtime errors.

Race conditions can be difficult to identify, replicate, and remediate because they are often caused by infinitesimal deviations in process timing.[iii] A SAS program that implements parallel processing can execute correctly the first 100 times, then suddenly and inexplicably fail on the 101st execution. Reliability and credibility of the program are compromised due to the failure, but this may overlooked because the program again begins functioning correctly on the 102nd execution and beyond. In essence, the failure is written off as a "fluke" and SAS practitioners may not perform the due diligence of investigating underlying logic errors or a lack of concurrency controls that abetted the race condition.

The stress testing performed in the TEST_COLLIDE macro was sufficient to demonstrate several failure patterns and the 100,000 iterations provide some confidence that all failure patterns were observed. However, because race conditions involve software timing issues, any internal or external factors that vary process execution time can exploit underlying race conditions. For example, because the PERM.Test data set is a consistent size, the file size would need to be varied to stress test the process more fully. The number of SAS sessions concurrently vying for access could also vary the intensity and exacerbate a race condition. Thus, concurrent processes might execute without failure when only two sessions are utilized, but adding a third or fourth session (also competing for exclusive access to the same data set) might exploit a previously unidentified race condition.

Because of the difficulty of identifying (and correcting) race conditions that lie camouflaged within seemingly innocuous software logic, the best defense against race conditions is to understand and avoid them in the first place. Through threat identification and defensive programming design, concurrency controls can be implemented that facilitate more secure and reliable parallel processing. For example, if only a shared lock is required (e.g., by the FREQ procedure), other procedures that also require a shared lock can execute without conflict. However, when a process requires an exclusive file lock in a multiuser or parallel processing environment, race conditions must be considered in software design and concurrency controls should validate file availability before attempted access.

STRESS TESTING CONTROL TABLES

The previous TEST_COLLIDE macro stress tests creation of a permanent data set (PERM.test) to elicit basic failure patterns that can occur due to file access collisions. However, because the aim of this text is to demonstrate methods that can be used to access control tables reliably from multiple SAS sessions, a subtle scenario variation is required. Control tables typically require bidirectional communication in which data are both read from and written to the control table, so this paradigm can be represented with the following code in which a data set is read, modified, and saved:

```
data perm.control;
    set perm.control;
    * perform transformations;
run;
```

The control table paradigm creates a more complex interaction in parallel processing design because the permanent data set is not only created (i.e., written) but also ingested (i.e., read). The DATA step requires an exclusive lock and the SET statement a shared lock, which work well in tandem when coded in a single, serialized program. However, if two SAS sessions concurrently execute the previous DATA step, the potential for collisions is amplified because of the additional pairings. For example, the DATA step in the first session (requiring an exclusive lock) could collide with either the DATA step or the SET statement in the second session, whereas in the TEST_COLLIDE macro, the DATA

step in the first session only had to contend with the DATA step in the second session. As demonstrated in Table 2, this subtlety will result in vastly more diverse and complex errors than those exhibited in Table 1.

To demonstrate the effect of this complexity on file collision failure patterns, the following stress test creates a control table (PERM.Control) and repeatedly accesses the data set from two SAS sessions via the TEST_CONTROL macro, simulating the type of rapid access that could occur when a control table is accessed repeatedly in parallel processing design. In the first SAS session, execute the following code, which creates a nine-observation data set and iteratively reads and updates it 1,000 times:

```
libname perm 'c:\perm'; * change to appropriate location;

data perm.control;
    length char $12;
    do i=1 to 9;
       char='SAS is rad!';
       output;
       end;
run;

%macro test_control(dsn=, cnt=, session=);
proc printto log="%sysfunc(pathname(perm))\log&session..log" new;
run;
%do i=1 %to &cnt;
    %put START: &i;
    data &dsn;
       set &dsn;
       * perform transformations;
    run;
    %put STOP;
    %end;
proc printto;
run;
%mend;

%test_control(dsn=perm.control, cnt=1000, session=1);
```

Immediately after starting the program, execute the following code in the second SAS session so that the programs are executing concurrently in both sessions:

```
libname perm 'c:\perm'; * change to appropriate location;

*** insert test_control macro definition here ***;

%test_control(dsn=perm.control, cnt=1000, session=2);
```

As before, the second session performs the same actions as the first but records the results in a separate log file to ensure the logs of each respective session are not overwritten. When the programs in both sessions have completed, the READLOG1 macro should be run in the first session to parse and analyze the log file from the first session:

```
%readlog1;
```

Table 2 demonstrates 16 unique failure patterns (shown in red) that occurred when two SAS sessions attempt to access a shared control table without appropriate concurrency controls. The frequency of specific runtime errors will differ each time the macros are executed, a common variability signature of race conditions. The DATA step succeeded in only 7.1 percent (N=71) of attempts (shown in black).

Log (Session One Only)	Frequency	Percent
*** NOTE: There were 0 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 0 observations and 0 variables. *** NOTE: DATA statement used (Total process time):	512	51.20
*** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** WARNING: The data set PERM.CONTROL was only partially opened and will not be saved. *** NOTE: DATA statement used (Total process time):	162	16.20
*** ERROR: A lock is not available for PERM.CONTROL.DATA. *** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** NOTE: DATA statement used (Total process time):	94	9.40
*** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** ERROR: File PERM.CONTROL.DATA is not open. *** WARNING: The data set PERM.CONTROL was only partially opened and will not be saved. *** NOTE: DATA statement used (Total process time):	88	8.80
*** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2 variables. *** NOTE: DATA statement used (Total process time):	71	7.10
*** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** WARNING: The data set PERM.CONTROL may be incomplete. When this step was stopped there were 0 *** observations and 0 variables. *** WARNING: Data set PERM.CONTROL was not replaced because this step was stopped. *** NOTE: DATA statement used (Total process time):	24	2.40
*** ERROR: File PERM.CONTROL.DATA does not exist. *** NOTE: The SAS System stopped processing this step because of errors. *** WARNING: The data set PERM.CONTROL may be incomplete. When this step was stopped there were 0 *** observations and 0 variables. *** WARNING: Data set PERM.CONTROL was not replaced because this step was stopped. *** NOTE: DATA statement used (Total process time):	17	1.70
*** ERROR: File PERM.CONTROL.DATA does not exist. *** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** NOTE: DATA statement used (Total process time):	12	1.20
*** ERROR: File PERM.CONTROL.DATA does not exist. *** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** WARNING: The data set PERM.CONTROL was only partially opened and will not be saved. *** NOTE: DATA statement used (Total process time):	5	0.50
*** NOTE: There were 0 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 0 observations and 0 variables. *** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: DATA statement used (Total process time):	5	0.50
*** ERROR: User does not have appropriate authorization level for file PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** WARNING: The data set PERM.CONTROL may be incomplete. When this step was stopped there were 0 *** observations and 0 variables. *** WARNING: Data set PERM.CONTROL was not replaced because this step was stopped. *** NOTE: DATA statement used (Total process time):	3	0.30
*** ERROR: File PERM.CONTROL.DATA does not exist. *** NOTE: The SAS System stopped processing this step because of errors. *** WARNING: The data set PERM.CONTROL may be incomplete. When this step was stopped there were 0 *** observations and 0 variables. *** NOTE: DATA statement used (Total process time):	2	0.20
*** ERROR: A lock is not available for PERM.CONTROL.DATA. *** ERROR: User does not have appropriate authorization level for file PERM.CONTROL.DATA. *** NOTE: The SAS	1	0.10

Log (Session One Only)	Frequency	Percent
System stopped processing this step because of errors. *** NOTE: DATA statement used (Total process time):		
*** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** WARNING: The data set PERM.CONTROL may be incomplete. When this step was stopped there were 0 *** observations and 0 variables. *** NOTE: DATA statement used (Total process time):	1	0.10
*** ERROR: User does not have appropriate authorization level for file PERM.CONTROL.DATA. *** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** NOTE: DATA statement used (Total process time):	1	0.10
*** NOTE: There were 0 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 0 observations and 0 variables. *** ERROR: Rename of temporary member for PERM.CONTROL.DATA failed. *** File can be found in c:\perm. *** NOTE: DATA statement used (Total process time):	1	0.10
*** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2 variables. *** ERROR: Rename of temporary member for PERM.CONTROL.DATA failed. *** File can be found in c:\perm. *** NOTE: DATA statement used (Total process time):	1	0.10

Table 2. Sample Frequency of 1,000 Iterations of TEST_CONTROL Macro

One of the most salient and disturbing aspects of the notes and runtime errors in Table 2 is the statement “0 observations and 0 variables” that is repeated across numerous failure paths. Attempting to open the PERM.Control data set confirms the horror that the control table was in fact corrupted and contains neither structure nor data after TEST_CONTROL completed. This alarming outcome indicates that it is possible to obliterate a SAS data set simply by introducing a race condition in which a separate process concurrently attempts to access the same data set, representing an unacceptable security risk to SAS software!

A single traumatic event led to the corruption of the data set, so it’s critical to understand what conditions preceded and precipitated this corruption. To further investigate this failure, open the two log files—Log1.log and Log2.log—in a text editor and search for the first occurrence of “0 observation” in each text file. The location of specific runtime errors will vary each time the programs are run, but in this example, the Log2 file includes the following output immediately preceding and including the first occurrence of “0 observation:”

```
START: 62

NOTE: There were 9 observations read from the data set PERM.CONTROL.
NOTE: The data set PERM.CONTROL has 9 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

STOP
START: 63
ERROR: File PERM.CONTROL.DATA does not exist.

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set PERM.CONTROL may be incomplete.  When this step was stopped
there were 0 observations and 0 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
```

```
      cpu time            0.01 seconds

STOP
```

If the data are successfully read in one iteration and have vanished by the next iteration, this points toward the second session as the victim—not the culprit—since no errors in the second session indicate the cause of file corruption. By similarly inspecting the Log1 file and tracing the output from the last occurrence of “9 observations” to the first occurrence of “0 observations,” the traumatic event can be easily identified:

```
START: 115

NOTE: There were 9 observations read from the data set PERM.CONTROL.
NOTE: The data set PERM.CONTROL has 9 observations and 2 variables.
ERROR: Rename of temporary member for PERM.CONTROL.DATA failed.
 File can be found in c:\perm.
NOTE: DATA statement used (Total process time):
      real time           1.02 seconds
      cpu time            0.18 seconds

STOP
START: 116
ERROR: A lock is not available for PERM.CONTROL.DATA.

ERROR: A lock is not available for PERM.CONTROL.DATA.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

STOP
START: 117

ERROR: A lock is not available for PERM.CONTROL.DATA.
NOTE: The SAS System stopped processing this step because of errors.
ERROR: File PERM.CONTROL.DATA is not open.
WARNING: The data set PERM.CONTROL was only partially opened and will not be saved.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

STOP
START: 118

ERROR: A lock is not available for PERM.CONTROL.DATA.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set PERM.CONTROL was only partially opened and will not be saved.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

STOP
START: 119

NOTE: There were 0 observations read from the data set PERM.CONTROL.
```

```
NOTE: The data set PERM.CONTROL has 0 observations and 0 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.03 seconds

STOP
```

In case you missed it, the 115th iteration caused the corruption, producing the following runtime error:

```
NOTE: There were 9 observations read from the data set PERM.CONTROL.
NOTE: The data set PERM.CONTROL has 9 observations and 2 variables.
ERROR: Rename of temporary member for PERM.CONTROL.DATA failed.
 File can be found in c:\perm.
```

Examination of Table 2 confirms this suspicion, indicating that this failure pattern occurred only once. Explanation of the root cause of this error is beyond the scope of this text, but isolation of the error confirms that it is possible to corrupt and delete a SAS data set simply by attempting to access the data set from a separate SAS session at the exact right moment. Moreover, because the race condition was revealed after only 115 iterations of TEST_CONTROL in session one and 62 iterations in session two, this indicates that the error likelihood is high. While these control tables contain bogus data, a control table just as easily could have been erased that included months of historic data, representing an enormous risk to multiuser SAS infrastructures or any software accessing data sets that reside in permanent libraries in networked environments. In fact, the LOCK statement was devised to combat this very risk but, as demonstrated in the following section, provides only a false sense of security.

STRESS TESTING THE LOCK STATEMENT

Given the description of the LOCK statement within SAS 9.3 documentation, implementation of LOCK (and subsequent dynamic testing of &SYSLCKRC) within the previous TEST_CONTROL macro should *in theory* eliminate the runtime errors demonstrated in Tables 1 and 2. Thus, by assessing that a data set is already locked (i.e., &SYSLCKRC is not equal to 0), the subsequent DATA step (that requires shared or exclusive access to the data set) can be dynamically circumvented to prevent file access collision. However, when TEST_CONTROL is modified to include the LOCK statement and &SYSLCKRC assessment, stress testing the TEST_LOCK macro demonstrates the inherent failure of LOCK, yielding unreliability that no doubt led to its deprecation within SAS 9.4 documentation.

To demonstrate stress testing and failure of the LOCK statement, execute the following code in the first SAS session, which creates a nine-observation control table and iteratively reads and updates it 1,000 times:

```
libname perm 'c:\perm'; * change to appropriate location;

data perm.control;
    length char $12;
    do i=1 to 9;
       char='SAS is rad!';
       output;
       end;
run;

%macro test_lock(dsn=, cnt=, session=);
proc printto log="%sysfunc(pathname(perm))\log&session..log" new;
run;
%do i=1 %to &cnt;
    %put START: &i;
    lock &dsn;
    %put SYSLCKRC: &syslckrc;
    %if &syslckrc=0 %then %do;
```

```
        data &dsn;
           set &dsn;
           * perform transformations;
        run;
        lock &dsn clear;
        %end;
    %put STOP;
    %end;
proc printto;
run;
%mend;

%test_lock(dsn=perm.control, cnt=1000, session=1);
```

Immediately after starting the program, execute the following code in the second SAS session so that the programs are executing concurrently in both sessions:

```
libname perm 'c:\perm'; * change to appropriate location;

*** insert test_lock macro definition here ****;

%test_lock(dsn=perm.control, cnt=1000, session=2);
```

The log analysis differs slightly from the READLOG1 macro because the READLOGS macro now combines both session logs into a single frequency table to ensure that all failure patterns occurring in either session are captured. When the programs in both sessions have completed, the following code should be run in the first session to parse and analyze the log files from both sessions:

```
%macro readlogs();
%do i=1 %to 2;
data perm.readlog&i (keep=obs log);
    length tab $200 obs 8 log $2000;
    infile "%sysfunc(pathname(perm))\log&i..log" truncover;
    input tab $200.;
    if _n_=1 then do;
       obs=0;
       log='';
       end;
    if substr(tab,1,6)='START:' then do;
       obs=substr(tab,7,12);
       log='';
       end;
    else if substr(tab,1,4)='STOP' then output;
    else do;
       if not(find(tab,"real time")>0 or find(tab,"cpu time")>0) and
             length(tab)>1 then do;
          log=strip(log) || ' *** ' || tab;
          end;
       end;
    retain log obs;
run;
%end;

data perm.readlog;
```

```
    set perm.readlog1 perm.readlog2;
run;

proc freq data=perm.readlog order=freq;
    tables log /nocum;
run;
%mend;

%readlogs;
```

Given SAS 9.3 documentation, the LOCK statement (and subsequent assessment of &SYSLCKRC) should prevent external SAS sessions from accessing an exclusively locked data set. Thus, in theory, only two possible outcomes should occur in each of the two sessions. For example, in the first session, the only two outcomes should include:

1. Session one attempts to lock the data set; the lock is gained (i.e., &SYSLCKRC is 0), and; the DATA step creates the PERM.Control data set without error.
2. Session one attempts to lock the data set; the data set is already locked by session two (i.e., &SYSLCKRC is not 0), and; the DATA step is dynamically skipped to prevent a file collision and failure.

However, because SAS 9.3 documentation grossly overstates the actual capabilities of the LOCK statement, numerous other failure patterns emerge when the LOCK statement is stress tested by executing the TEST_LOCK macro across two SAS sessions. Moreover, because some of these runtime errors are extremely rare, the DATA step within the TEST_LOCK macro is iterated 100,000 times to facilitate capture of uncommon failure patterns. Table 3 demonstrates the frequency of success and failure patterns that emerge when LOCK is implemented, with failures caused by deficiencies in the LOCK statement shown in red.

Log (Session One and Two Combined)	Frequency	Percent
*** ERROR: A lock is not available for PERM.CONTROL.DATA. *** SYSLCKRC: 70031	192,074	96.04
*** NOTE: PERM.CONTROL.DATA is now locked for exclusive access by you. *** SYSLCKRC: 0 *** ERROR: Permanent copy of file PERM.CONTROL.DATA was deleted. *** NOTE: The SAS System stopped processing this step because of errors. *** ERROR: File PERM.CONTROL.DATA is not open. *** NOTE: DATA statement used (Total process time): *** NOTE: PERM.CONTROL.DATA is no longer locked by you.	5,426	2.71
*** NOTE: PERM.CONTROL.DATA is now locked for exclusive access by you. *** SYSLCKRC: 0 *** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2 variables. *** NOTE: DATA statement used (Total process time): *** NOTE: PERM.CONTROL.DATA is no longer locked by you.	1,149	0.57
*** NOTE: PERM.CONTROL.DATA is now locked for exclusive access by you. *** SYSLCKRC: 0 *** ERROR: File PERM.CONTROL.DATA is damaged. I/O processing did not complete. *** NOTE: The SAS System stopped processing this step because of errors. *** ERROR: File PERM.CONTROL.DATA is not open. *** NOTE: DATA statement used (Total process time): *** NOTE: PERM.CONTROL.DATA is no longer locked by you.	547	0.27
*** ERROR: You cannot lock PERM.CONTROL.DATA with a LOCK statement because PERM.CONTROL.DATA does not *** exist. *** SYSLCKRC: 630289	437	0.22
*** NOTE: PERM.CONTROL.DATA is now locked for exclusive access by you. *** SYSLCKRC: 0 *** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: The SAS System stopped processing this step because of errors. *** ERROR: File PERM.CONTROL.DATA is not open. *** WARNING: The data set PERM.CONTROL was only partially opened and will not be saved. *** NOTE: DATA statement used (Total process time): *** NOTE: PERM.CONTROL.DATA is no longer locked by you.	296	0.15
*** NOTE: PERM.CONTROL.DATA is now locked for exclusive access by you. ***	56	0.03

Log (Session One and Two Combined)	Frequency	Percent
SYSLCKRC: 0 *** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2 variables. *** ERROR: A lock is not available for PERM.CONTROL.DATA. *** NOTE: DATA statement used (Total process time): *** NOTE: PERM.CONTROL.DATA is no longer locked by you.		
*** ERROR: User does not have appropriate authorization level for file PERM.CONTROL.DATA. *** SYSLCKRC: 70030	15	0.01

Table 3. Sample Frequency of 100,000 Iterations of TEST_LOCK Macro

In 96.0 percent (N=192,074) of the iterations, an exclusive lock could not be obtained but this should not be considered a failure so it is shown in black. Although "ERROR" is printed to the SAS log, this represents an exception rather than a runtime error because the &SYSLCKRC automatic macro variable correctly assesses that a lock was not achieved (70031). Thus, in the vast majority of iterations, LOCK performed as expected and advertised. Although the data set was successfully locked and accessed in only 0.57 percent (N=1,149) of the iterations, a low access percentage is expected because more resources are consumed when executing a DATA step than when testing a LOCK. For example, if the first session gains a lock on the data set and executes the DATA step, the second session might incur fifty or more failed lock attempts while session one maintains the lock. The "User does not have appropriate authorization level" error is somewhat enigmatic and outside the scope of this text, but because &SYSLCKRC again accurately depicts an unsuccessful LOCK statement (70030), the LOCK statement performs admirably in these cases, thus they are also highlighted in black.

All other iterations (shown in red) depict actual failure of the LOCK statement—that is, not just the exception of an unsuccessful LOCK statement, but the production of spurious results. The traumatic failure that causes corruption of the control table appears to have been eliminated (based on the lack of "0 observations" references in the log), but a new troubling pattern has emerged: The &SYSLCKRC automatic macro variable is repeatedly assessed to be 0 (indicating a lock has been achieved) immediately after which an error stating that "A lock is not available" is produced. This alarming contradiction should never exist.

The various outcomes are worth inspection because any successful solution that enables secure file locking will need to overcome the inherent deficiencies of the LOCK statement. The outcomes include:

1. **LOCK prevents a file access collision** — The LOCK statement correctly assesses the data set was in use and prevents file access collisions. Again, despite the Base SAS nuance that prints "ERROR" to the log, the LOCK statement functions correctly in these cases.
2. **A lock is gained** — The LOCK statement succeeds (&SYSLCKRC is 0) and the control table is updated.
3. **LOCK lacks permission** — The least common "error" occurs during only 0.05 percent (N=15) of iterations and is not actually a failure of the LOCK statement, but rather an eccentricity of Base SAS. Because &SYSLCKRC correctly identifies that a lock could not be achieved (70030), subsequent code that would have required a lock can be dynamically rerouted through exception handling and programmatic assessment of &SYSLCKRC.
4. **LOCK fails but lies to users and states it succeeded** — In approximately 0.45 percent (N=899) of iterations, the LOCK statement blusters that "PERM.CONTROL.DATA is now locked for exclusive access by you" which is untrue. In these unfortunate cases, the &SYSLCKRC value is set to 0, also falsely touting LOCK success.
5. **LOCK lies and says the data set has been deleted** — Despite the runtime error indicating that the "PERM.CONTROL.DATA does not exist," this result is again nothing more than the LOCK statement telling fibs. Because the PERM.Control data set was created only once in the first session, if it exists at the close of both sessions, this demonstrates the data set was not deleted. Examination of PERM.Control following completion of the TEST_CONTROL macro confirms that "SAS is [still] rad!" and signals that no corruption occurred.

Outcomes four and five—the two failure paths—are caused by a failure of LOCK to identify that data sets are in use. Although the full explanation lies outside the scope of this text, the first failure results in part because LOCK fails to

assess the temporary utility file that is created when SAS builds a data set. For example, when the PERM.Control data set (C:\perm\control.sas7bdat) is created, a temporary utility file (C:\perm\control.sas7bdat.lck) is created while the data set is constructed. Only after completion is the utility file renamed the actual SAS data set. Thus, because of underlying race conditions within the Base SAS language, LOCK mistakenly believes it has achieved an exclusive lock on the data set when in fact the subsequent DATA step will fail because the utility file already exists and is exclusively locked.

To demonstrate this deficiency, an LCK utility file can be created manually, after which LOCK will still think (and report) that it has successfully locked the corresponding data set—when in fact it has not. For example, when C:\perm\test.sas7bdat.lck is locked by session one, no other SAS session should be able to gain an exclusive lock on C:\perm\test.sas7bdat (PERM.Test). To demonstrate this concurrency failure of the LOCK statement, execute the following code in the first SAS session, which creates the PERM.Test data set, waits three seconds (during which the LCK file is unlocked), locks the LCK file for three seconds, and finally unlocks the LCK file and waits three seconds:

```
%macro test_lck_lock1();
data perm.test;
    length char1 $10;
    char1='lock me up';
run;
%do i=1 %to 3;
    %put &i unlocked;
    %let sleeping=%sysfunc(sleep(1));
    %end;
%let loc=%sysfunc(pathname(perm))\test.sas7bdat.lck;
filename myfil "&loc";
%let dsid=%sysfunc(fopen(myfil,u));
%do i=1 %to 3;
    %put &i locked;
    %let sleeping=%sysfunc(sleep(1));
    %end;
%let closed=%sysfunc(fclose(&dsid));
%do i=1 %to 3;
    %put &i unlocked;
    %let sleeping=%sysfunc(sleep(1));
    %end;
%mend;

%test_lck_lock1;
```

The FOPEN function with the “U” (Update) parameter directs that a read-write stream be opened to the file reference (MYFIL) which requires an exclusive lock. By specifying the C:\perm\test.sas7bdat.lck file reference in FOPEN, the LCK file is created and locked manually. The corresponding FCLOSE function is required to close the data stream and release the exclusive lock so that other users or processes can again access the file. Under normal SAS operation, the creation, locking, unlocking, and deletion of LCK utility files occur automatically and discreetly, but this novel approach recreates the underlying mechanics to investigate inconsistencies in the LOCK statement.

Immediately after starting the program, execute the following code in the second SAS session so that the programs are executing concurrently in both sessions:

```
%macro test_lck_lock2();
%do i=1 %to 9;
    lock perm.test;
    lock perm.test clear;
    %let sleeping=%sysfunc(sleep(1));
```

```
    %end;
%mend;

%test_lck_lock2;
```

The code in the second session tests the lock status of the PERM.Test data set once per second, and were LOCK integrated with the utility LCK functionality, it would assess that a lock is not available for the three middle iterations (because the first session has exclusively locked the corresponding LCK utility file). The output from the first session demonstrates the lock held on the LCK file during the three middle iterations:

```
1 unlocked
2 unlocked
3 unlocked
1 locked
2 locked
3 locked
1 unlocked
2 unlocked
3 unlocked
```

However, the log from the second session indicates that LOCK failed to recognize that the LCK file was exclusively locked, reporting a successful lock of the PERM.Test data set for all nine iterations:

```
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
NOTE: PERM.TEST.DATA is now locked for exclusive access by you.
NOTE: PERM.TEST.DATA is no longer locked by you.
```

LOCK reports that an exclusive lock was obtained by the second session but, for the three middle iterations, the second session would have been unable to access PERM.Test because its corresponding LCK file was exclusively locked. The deprecation of the LOCK statement due to this and other inconsistencies is discussed in the following section.

THE SAD DEMISE OF LOCK

Like an aging Hollywood starlet, poor old LOCK died quietly and alone. Table 4 demonstrates a comparison between SAS 9.3 and SAS 9.4 documentation, indicating that SAS either dramatically curtailed the functionality of LOCK in SAS 9.4 or realized it was broken and stopped promoting it as the parallel processing panacea. LOCK capabilities alleged in SAS 9.3 but removed in SAS 9.4 are shown in bold red, as are usage caveats added in SAS 9.4 but absent in SAS 9.3.

SAS® 9.3 Statements: Reference LOCK Statement[iv]	SAS® 9.4 Statements: Reference, Fourth Edition LOCK Statement[v]
Primary Use of Lock	
"The LOCK statement enables you to acquire and release an exclusive lock on an existing SAS file. **Once an exclusive lock is obtained, no other SAS session can read or write to the file until the lock is released."**	"The LOCK statement enables you to acquire, list, or release an exclusive lock on an existing SAS file. With an exclusive lock, no other operation **in the current SAS session** can read, write, or lock the file until the lock is released."
First Example	
"For example, consider a nightly update process that consists of a DATA step to remove observations that are no longer useful, a SORT procedure to sort the file, and a DATASETS procedure to rebuild the file's indexes. If another SAS session accesses the file between any of the steps, the SORT and DATASETS procedures would fail, because they require member-level locking (exclusive) access to the file. **Including the LOCK statement before the DATA step provides the needed protection by acquiring exclusive access to the file. If the LOCK statement is successful, a SAS session that attempts to access the file between steps will be denied access, and the nightly update process runs uninterrupted."**	"The primary use of the LOCK statement is to retain exclusive control of a SAS file across SAS statement boundaries. There are times when it is desirable to perform several operations on a file, one right after the other, without losing exclusive control of the file. However, when a DATA or PROC step finishes executing and control flows from it to the next operation, the file is closed and becomes available for processing by another SAS session that has access to the same data storage location. **To ensure that an exclusive lock is guaranteed across multiple SAS sessions, you must use SAS/SHARE."**
Second Example	
"The following SAS program illustrates the process of locking a SAS data set. Including the LOCK statement provides protection for the multistep program by acquiring exclusive access to the file. **Any SAS session that attempts to access the file between steps will be denied access, which ensures that the program runs uninterrupted."**	"The following SAS program illustrates the process of locking a SAS data set. Including the LOCK statement provides protection for the multistep program by acquiring exclusive access to the file."
Second Example Description	
"1. Acquires exclusive access to the SAS data set MYDATA.CENSUS. 2. Opens MYDATA.CENSUS to remove observations that are no longer useful. At the end of the DATA step, the file is closed. However, because of the exclusive lock, any other SAS session that attempts to access the file is denied access. 3. Opens MYDATA.CENSUS to sort the file. **At the end of the procedure, the file is closed but not available to another SAS session.** 4. Opens MYDATA.CENSUS to rebuild the file's index. At the end of the procedure, **the file is closed but still not available to another SAS session.** 5. Releases the exclusive lock on MYDATA.CENSUS. The data set is now available to other SAS sessions."	"1. Acquires exclusive access to the SAS data set MyData.Census. 2. Opens MyData.Census to remove observations. During the update, no other operation in the current SAS session and no other SAS session can access the file. 3. At the end of the DATA step, the file is closed. No other operation in the current SAS session can access the file. **However, until the file is reopened, it can be accessed by another SAS session.** 4. Opens MyData.Census to sort the file. During the sort, no other operation in the current SAS session and no other SAS session can access the file. At the end of the procedure, **the file is closed**, which means that no other operation in the current SAS session can access the file **but it can be accessed by another SAS session.** 5. Opens MyData.Census to rebuild the file's index. No other operation in the current SAS session and no other SAS session can access the file. At the end of the procedure, the file is closed. 6. Releases the exclusive lock on MyData.Census."

Table 4. Discrepancies Between LOCK Statement Capabilities in SAS 9.3 and 9.4

Most notably, in the second example in LOCK statement documentation, Table 4 indicates that while the code is identical between SAS 9.3 and 9.4 documentation, the description in SAS 9.4 eliminates the security that LOCK purportedly provided in SAS 9.3 and adds the caveat that the data set "can be accessed by another SAS session" despite being locked. Yet SAS 9.3 documentation proclaims "Once an exclusive lock is obtained, no other SAS session can read or write to the file until the lock is released." So Base SAS may *create* a lock…but not *hold* the lock…reminiscent of the classic exchange between Jerry Seinfeld and an airport rental car agent in the Seinfeld's third-season episode *The Alternate Side*:[vi]

> Agent: Well I'm sorry, we have no mid-size available at the moment.
>
> Jerry: I don't understand—I made a reservation. Do you have my reservation?
>
> Agent: Yes, we do. Unfortunately we ran out of cars.
>
> Jerry: But the reservation keeps the car here. That's why you have the reservations.
>
> Agent: I know why we have reservations.
>
> Jerry: I don't think you do. If you did, I'd have a car. See, you know how to *take* the reservation. You just don't know how to *hold* the reservation. And that's really the most important part of the reservation—the *holding*. Anybody can just take them.
>
> Agent: Let me speak with my supervisor… Would you like insurance?
>
> Jerry: Yeah, you better give me the insurance because I am going to beat the hell out of this thing.

MUTEX SEMAPHORES SAVE THE DAY

To facilitate reliable, secure, exclusive data set access, several criteria must be met. A SAS process must be able to:

1. determine programmatically when a data set is locked (i.e., in use) to reroute program flow away from the locked data set to avoid conflict
2. determine programmatically when a data set is unlocked (i.e., not in use) and lock the data set exclusively
3. signal to other users and processes that it has locked a data exclusively
4. lock a data set securely so that no external process can cause process failure
5. unlock a data set securely and efficiently when exclusive access is no longer required
6. signal to other users and processes that it has unlocked a data set

SAS 9.3 documentation alleges that the LOCK statement fulfills this laundry list of requirements, but because LOCK falls short, an alternative methodology is required to supplant the LOCK statement to facilitate concurrent control table use in parallel processing design. Conceptually, the logic within the implementation of LOCK demonstrated in the TEST_LOCK macro is correct, which incorporates two methods common in multithreaded and parallel design—the semaphore and mutex. A *semaphore* is "a shared variable used to synchronize concurrent processes by indicating whether an action has been completed or an event has occurred."[vii] Semaphores are flags that represent software events, conditions, or attributes, such as file lock status. Whereas &SYSLCKRC represents the return code (i.e., successful or failed) of the most recent LOCK statement, a semaphore should always be present and perfectly correlated with the object or state it represents. Thus, whereas &SYSLCKRC represents only the most recent attempt to lock a data set, a separate semaphore would be required to signal lock status for each data set in a shared environment.[viii]

A *mutex semaphore* (or *mutex*) represents a class of semaphore that not only flags lock status but also ensures mutually exclusive access to some object, and is defined as "a mechanism for implementing mutual exclusion."[ix] When a semaphore is implemented to represent file lock status (i.e., data set availability), it flags the state to alert other processes. However, the semaphore alone does nothing to restrict access to the data set from other processes, so it provides no security. For example, some semaphores are used to count or limit the number of threads (or processes) accessing some object concurrently, but don't restrict access to one party. Thus, mutex semaphores take

this additional step to provide secure and reliable *exclusive* file access. While *mutex* denotes a type of semaphore, the term can also refer to the actual business logic that enforces mutual exclusion, thus representing the combination of the flag and the concurrency control.

Recall that in OOP languages, race conditions often occur in multithreaded processing when separate threads within the same program race to lock or control some object. Because the communication occurs within the same program, variables are typically used as semaphores to signal object availability. Race conditions that plague SAS parallel processing, however, typically occur because processes in separate SAS sessions are racing to access or lock the same data set, log file, or output file. Because SAS variables and macro variables cannot be passed between SAS sessions (outside of complex methods that implement RSUBMIT or SYSPARM), text files represent one method to facilitate inter-session communication and can be created with the FOPEN statement.

If the PERM.Control data set is intended to be accessed concurrently by two or more SAS sessions, a mutex must be dynamically assigned to represent the data set uniquely. The text file C:\perm\control.txt.lck is created (or exclusively accessed if it already exists) in the following code:

```
libname perm 'c:\perm'; * change to appropriate location;
%let dsn=control.perm;
%let lib=%scan(&dsn,1);
%let tab=%scan(&dsn,2);
%let loc=%sysfunc(pathname(&lib))\&tab..txt.lck;
filename myfil "&loc";
%let dsid=%sysfunc(fopen(myfil,u));
```

If FOPEN succeeds, a read-write data stream to the text file is opened so an exclusive lock is created for Control.txt.lck. If FOPEN fails, however, this indicates that a separate session already has an exclusive lock on the text file, which can be programmatically tested because &DSID will be set to 0. The lock status of the text file thus is the semaphore representing the lock status of the associated data set. Moreover, the logic of the mutex tests &DSID and specifies that only when the text file is unlocked can its associated data set be accessed, thus preventing file access collisions. Stress testing of this mutex design is demonstrated in the following section.

STRESS TESTING A MUTEX SEMAPHORE SOLUTION

In the first SAS session, execute the following code, which creates a nine-observation data set and iteratively reads and updates it 100,000 times:

```
libname perm 'c:\perm'; * change to appropriate location;

data perm.control;
    length char $12;
    do i=1 to 9;
       char='SAS is rad!';
       output;
       end;
run;

%macro test_mutex(dsn=, cnt=, session=);
proc printto log="%sysfunc(pathname(perm))\log&session..log" new;
run;
%let lib=%scan(&dsn,1);
%let tab=%scan(&dsn,2);
%let loc=%sysfunc(pathname(&lib))\&tab..txt.lck;
%put LOC: &loc;
filename myfil "&loc";
%put SYSFILRC: &sysfilrc;
```

```
%do i=1 %to &cnt;
    %put START: &i;
    %put SESSION &session;
    %let dsid=%sysfunc(fopen(myfil,u));
    %put DSID: &dsid;
    %if &dsid>0 %then %do;
       %if %sysfunc(exist(&dsn)) %then %do;
          data &dsn;
             set &dsn;
             * perform transformations;
          run;
          %end;
       %let closed=%sysfunc(fclose(&dsid));
       %end;
    %put STOP;
    %end;
proc printto;
run;
%mend;

%test_mutex(dsn=perm.control, cnt=100000, session=1);
```

Immediately after starting the program, execute the following code in the second SAS session so that the programs are executing concurrently in both sessions:

```
libname perm 'c:\perm'; * change to appropriate location;

*** insert test_mutex macro definition here ***;

%test_mutex(dsn=perm.control, cnt=100000, session=2);
```

The log file analysis again combines both session logs into a single frequency table to ensure that all failure patterns occurring in either session are captured. When the programs in both sessions have completed, the READLOGS macro should be run in the first session to parse and analyze the log files from both sessions:

```
%readlogs;
```

Table 5 demonstrates results from the TEST_MUTEX macro when the DATA step is iterated 100,000 times in two concurrent SAS sessions. Although the failure patterns have been reduced in quantity and complexity, the familiar catastrophic “Rename of temporary member” error is observed, which corrupts or deletes the data set.

Log (Session One and Two Combined)	Frequency	Percent
*** SESSION 1 *** DSID: 0	68156	34.08
*** SESSION 2 *** DSID: 0	66735	33.37
*** SESSION 2 *** DSID: 1	32434	16.22
*** SESSION 1 *** DSID: 1	31009	15.50
*** SESSION 1 *** DSID: 1 *** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2 variables. *** NOTE: DATA statement used (Total process time):	835	0.42
*** SESSION 2 *** DSID: 1 *** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2	830	0.42

Log (Session One and Two Combined)	Frequency	Percent
variables. *** NOTE: DATA statement used (Total process time):		
*** SESSION 2 *** DSID: 1 *** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2 variables. *** ERROR: Rename of temporary member for PERM.CONTROL.DATA failed. *** File can be found in c:\perm. *** NOTE: DATA statement used (Total process time):	1	0.00

Table 5. Sample Frequency of 100,000 Iterations of TEST_MUTEX Macro

The “Rename of temporary member” failure indicates that in some cases the FOPEN statement itself also fails and produces spurious results in a concurrent processing environment. The “U” attribute of the FOPEN statement is intended to gain an exclusive lock on a text file (required for read-write access), and once that lock is in place, no other process should be able to lock or access the same text file. Thus, because the conditional logic specifies that the DATA step should execute only after confirmation of the lock of the associated text file (when &DSID > 0), somehow two DATA steps were still able to collide, indicating a failure of FOPEN to obtain an exclusive lock when it reported via return code that it had. The failure of FOPEN is not discussed further in this text, but warrants additional scrutiny as it exhibits race conditions similar to LOCK in concurrent processing environments.

The bizarre results again demonstrate the importance of stress testing software. Despite a promising solution that logically should have facilitated secure file locking, the methodology fails because of undocumented vulnerabilities in Base SAS software. Despite the results in Table 5, the failure pattern is more straightforward than previous ones and, with some subtle modification, can yield a more reliable solution. Because race conditions are inherently caused by slight deviations in the timing of software processes or threads, by adjusting the length of time a process takes to complete (or time between processes), race conditions can often be eliminated or their frequency substantially reduced. The difficulty in mitigating race conditions with timing manipulation is that it can be impossible to demonstrate that the risk has been eliminated, as opposed to merely reduced. For example, mitigating a race condition by adding a one second pause might reduce the likelihood of a file collision to one in one million; however, if that millionth collision causes a “Rename of temporary member” error and destruction of a critical control table, even this low likelihood could represent an unacceptable risk and sufficient rationale to not implement the solution.

To demonstrate one timing solution, the two TEST_MUTEX macros from this section are modified by inserting the SLEEP macro (which calls the SLEEP function in Windows and CALL SLEEP function in UNIX) immediately before the DATA step, causing a one second delay:

```
%if %sysfunc(exist(&dsn)) %then %do;
    %sleep(sec=1); * inserted code;
    data &dsn;
```

Note that because the SLEEP function is not portable between Windows and UNIX environments, the SLEEP macro (see Appendix A) is utilized to facilitate portability. When the TEST_MUTEX macros in both sessions are updated with this code and executed concurrently, the results demonstrate a possible elimination of the risk. As Table 6 demonstrates, when the READLOGS macro is run after the TEST_MUTEX macros have completed, the mutex now performs perfectly across 100,000 iterations.

Log (Session One and Two Combined)	Frequency	Percent
*** SESSION 2 *** DSID: 0	52516	26.26
*** SESSION 2 *** DSID: 1 *** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2 variables. *** NOTE: DATA statement used (Total process time):	47484	23.74
*** SESSION 1 *** DSID: 0	51322	25.66

Log (Session One and Two Combined)	Frequency	Percent
*** SESSION 1 *** DSID: 1 *** NOTE: There were 9 observations read from the data set PERM.CONTROL. *** NOTE: The data set PERM.CONTROL has 9 observations and 2 variables. *** NOTE: DATA statement used (Total process time):	48678	23.34

Table 6. Sample Frequency of 100,000 Iterations of TEST_MUTEX Macro with Delay

Despite a clean log, the point must be reiterated that it's difficult if not impossible to assess the risk or reliability of this solution, because solving a race condition by introducing a delay is difficult to test or logically validate. While the code performed admirably on *this* computer and *this* infrastructure and *this* SAS version with *this* sample control table, it's impossible to understand how these factors known to affect software speed and performance might interact to influence the race condition. Thus, while the stress test indicates that the race condition has been mitigated, it cannot be demonstrated that the race condition has been eliminated.

THE LOCKSAFE SOLUTION

The LOCKSAFE macro is included in Appendix A. The race condition demonstrated in the previous mutex solution can be eliminated by adding logic that identifies all sessions interested in a specific data set and subsequently identifies the session that first requested the data set. The macro requests data set access by creating a semaphore—a text file tagged not only with the data set name but also with the &SYSJOBID that uniquely identifies the session. In so doing, there is no chance that a separate session could simultaneously be attempting to create or update the semaphore (because the session IDs would differ) so file access collisions cannot occur between semaphores. After waiting a second, the macro then identifies all semaphores for the data set—including those created in concurrent SAS sessions—and selects the first (i.e., oldest) semaphore and awards that session the prize, allowing the session to lock the data set.

The one-second delay is required because it ensures that other current sessions also requesting access will be identified through their respective semaphores. Thus, if no mutex exists and two SAS sessions produce semaphores at the exact same instant requesting access to the same data set, the pause ensures that the semaphores are not only produced but also can be read and prioritized. Without the one-second delay, the two sessions could independently create semaphores and might independently assess that no other semaphore existed, thus causing a failure when both sessions were simultaneously granted exclusive access to the data set.

When the data set is locked, a mutex (i.e., empty text file) is also created that uniquely identifies the data set. The mutex is required because the semaphores only prioritize data set access to competing processes, whereas the mutex prevents subsequent semaphores from being formed. Thus, mutexes are beneficial because testing a mutex can immediately demonstrate that a data set is locked by another session. In demonstrating an existing file lock with a mutex, LOCKSAFE conserves system resources because it enters a busy-waiting cycle (and temporarily sleeps) without having to search for and analyze all semaphores to determine which first requested file access. Thus, if either a mutex exists or a mutex does not exist but another SAS session has already called dibs via a semaphore, subsequent SAS sessions will sleep for a parameterized amount of time. The macro exits when either a data set lock is achieved or the process times out. Figure 1 demonstrates the LOCKSAFE process flow.

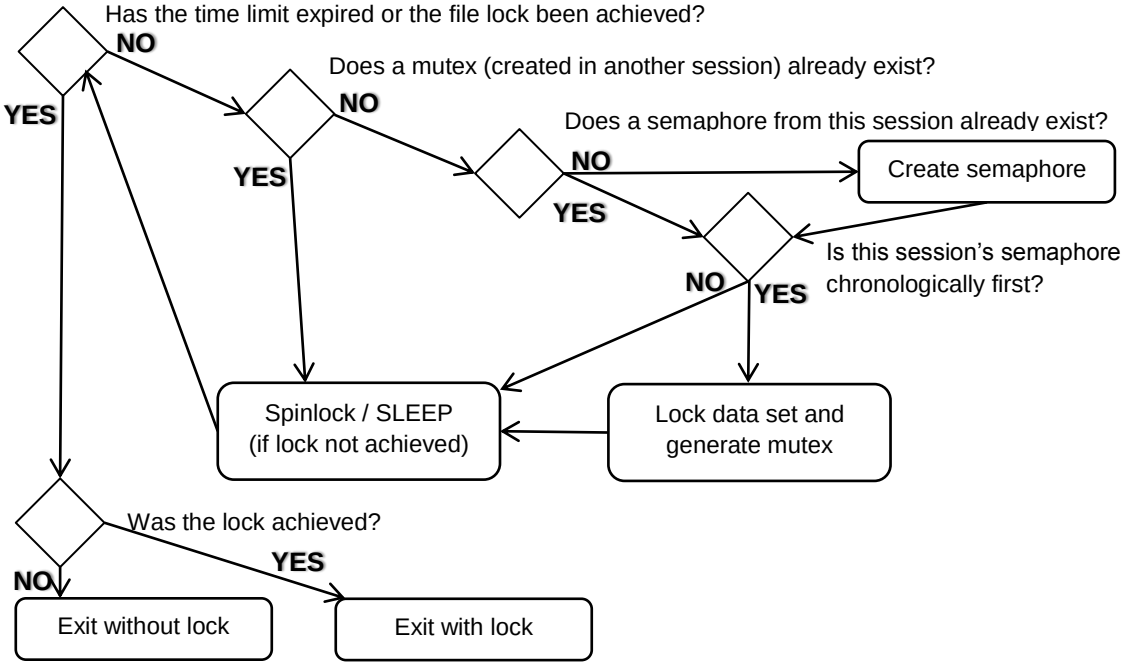

Figure 1. LOCKSAFE Process Flow

The LOCKSAFE macro is used both to lock and unlock a data set and its definition includes the following parameters:

```
%macro locksafe(dsn= /* data set name in LIB.DSN format */,
    sec=1 /* seconds to wait (1 or greater) between lock reattempts */,
    max=60 /* maximum number of seconds to attempt lock until process timeout */,
    seccollect=1 /* seconds to wait (1 or greater) to collect all processes */,
    secoffset=0 /* seconds (to three decimal places) to add or subtract */,
    terminate=NO /* YES to unlock rather than lock a data set */);
```

For example, the following initial invocation will attempt to lock the PERM.Control data set, but if another SAS session has already locked the data set using LOCKSAFE, this invocation will repeatedly test the availability of the data set every five seconds until the data set is accessible or the process times out in 10 minutes (i.e., 600 seconds):

```
%locksafe(dsn=perm.control, sec=5, max=600);
```

When the macro first tests the accessibility of the data set, it produces a semaphore file in the following format: LIB_DSN_SASDate_SECParameter_&SYSJOBID_.semaphore. For example, the previous LOCKSAFE invocation might produce the following semaphore file in the MUTEX library:

```
perm_control_1788981171.839_5_14935_.semaphore
```

When the SAS date in a semaphore file is earlier than the dates in other semaphore files, the session corresponding to that date subsequently produces the mutex file in the following format: LIB_DSN.mutex. For example, regardless of which SAS session first gained access to the data set, the previous LOCKSAFE invocation would produce the following mutex file in the MUTEX library:

```
perm_control.mutex
```

The optional SECCOLLECT parameter specifies the number of seconds that LOCKSAFE waits and collects semaphores from all processes to prioritize which process should secure the lock. The one-second default will work

in most environments and should be increased only if high network latency is present. The optional SECOFFSET parameter specifies the number of seconds (and thousandths of a second) that should be added to the current time. For example, because LOCKSAFE can be used to communicate between multiple sessions or instances of SAS, separate sessions running on separate processors might display different times. Even a split-second disparity in clock time could cause two concurrent sessions to believe they each were first in line for a data set, thus causing a file access collision. For this reason, in some cases, time must be added (or subtracted) from one or more sessions. For example, one and one-quarter second could be added by setting SECOFFSET to 1.25.

Once the data set has been accessed and a lock is no longer required, the data set must be unlocked, which only requires the DSN and TERMINATE parameters. For example, the following code unlocks the data set after use:

```
%locksafe(dsn=perm.control, terminate=YES);
```

Unlocking occurs in two steps. First, the mutex is deleted. Next, all semaphores that correspond to the session (identified by &SYSJOBID) are deleted, which allows other SAS sessions (implementing LOCKSAFE) to lock the data set. LOCKSAFE generates a single return code, the global macro variable LOCKSAFERC. An empty value signifies that LOCKSAFE did not encounter any exceptions or errors, while additional values include:

- **LOCKSAFE FAILURE —** This default return code value is initialized at macro inception and is produced when an unspecified warning or runtime error occurs (i.e., when &SYSCC is greater than 0).
- **FDELETE of Mutex Failed —** The FDELETE function is required to delete the mutex and semaphore files. Because the mutex file is also queried for existence by the FEXIST function (which requires a shared lock on the mutex), a race condition exists because FDELETE will fail if FEXIST is executing in a different session. To mitigate the race condition, FDELETE is repeated up to 1,000 times but, because an infinitesimal risk of failure of FDELETE still exists, the risk is captured in this failure path and return code.
- **LOCKSAFE Timeout After &MAX Seconds —** The lock could not be achieved before the maximum time threshold (MAX parameter) was reached. While the previous return codes indicate failures of the LOCKSAFE macro itself or of the environment (such as a missing data set), this return code is an expected exception that will occur when a data set is in use elsewhere. Thus, while encountering this exception in some circumstances will still necessitate process termination (as demonstrated in the following example), in other instances a timeout only signifies that program flow must dynamically shift to some unrelated task that does not require exclusive access to the specific data set.

LOCKSAFE should always be used within an exception handling framework, and the diversity and comprehensiveness of the LOCKSAFE return codes facilitate this objective. In a common LOCKSAFE use scenario, the control table PERM.Control might contain data that need to be updated by processes running concurrently in two or more SAS sessions. Each SAS session would need to execute LOCKSAFE before accessing the data set and would conditionally access the data set only when &LOCKSAFERC was blank, indicating a successful lock. For example, the following code terminates the macro if the lock cannot be achieved, thus avoiding possible corruption of the control table and other runtime errors resulting from file access collision:

```
%macro test;
%locksafe(dsn=perm.control, sec=5, max=120);
%if %length(&locksafeRC)=0 %then %do;
    data perm.control;
       set perm.control;
       * perform transformations;
    run;
    %locksafe(dsn=perm.control, terminate=YES);
    %if %length(&locksafeRC)^=0 %then %return;
    %end;
%else %return;
* do something else;
%mend;
```

```
%test;
```

Stress testing LOCKSAFE 100,000 times running on four concurrent SAS sessions produced perfect results. Because LOCKSAFE requires at least a one-second delay between iterations (due to the semaphore analysis process), stress testing LOCKSAFE is considerably slower than the previously demonstrated tests and ran for several consecutive days. The LOCKSAFE macro is designed to be included with the %INCLUDE statement or implemented through the Autocall Macro Facility. Because the LOCKSAFE macro requires the MUTEX library to be created to store temporary semaphore and mutex files, the following LIBNAME statement must be modified before LOCKSAFE invocation, such as this initialization in the SAS University Edition:

```
%let mutlib=mutex;
libname &mutlib '/folders/myfolders/mutex'; * MUST CHANGE TO ACTUAL LOCATION;
```

In the SAS Display Manager for Windows, the previous initialization might instead appear as:

```
%let mutlib=mutex;
libname &mutlib 'c:\perm\mutex'; * MUST CHANGE TO ACTUAL LOCATION;
```

With the MUTEX library named and its logical location specified, LOCKSAFE is ready to be implemented.

LOCKSAFE MACRO CAVEATS

The LOCKSAFE macro eliminates errors that plague the LOCK statement, enabling two or more processes running in separate SAS sessions to communicate with each other and synchronize access to a shared control table. To be clear, however, LOCKSAFE does not replace all LOCK functionality alleged in SAS 9.3 documentation. For example, the LOCK statement was designed to protect data sets from unplanned external access, thus a process *in theory* should have been able to implement LOCK to ensure that exclusive access was achieved and that the threat of rogue users or processes was eliminated. Because both FOPEN and LOCK fail in concurrent environments, however, LOCKSAFE cannot offer the protection once ascribed to LOCK. LOCKSAFE does facilitate planned communication and synchronization between concurrent SAS sessions sufficient to safely lock and unlock shared data sets.

The LOCKSAFE macro synchronizes concurrent processing, ensuring that all sessions that utilize LOCKSAFE to test and achieve data access will be successful or reliably time out in a controlled fashion. Thus, when an infrastructure successfully adopts and implements LOCKSAFE for use with control tables and other shared data sets, everyone has a nice day. But it only takes one rogue process attempting to access a data set (without LOCKSAFE) to unhinge this reliability. For example, while the previous LOCKSAFE example could be executed concurrently on one or one hundred SAS sessions with confidence, if a separate session ran a process that explicitly or implicitly (i.e., with or without the LOCK statement, respectively) gained a shared or exclusive lock on the same data set, this would cause LOCKSAFE to fail. Thus, while LOCKSAFE takes tremendous strides forward in its ability to facilitate communication via control tables in parallel processing design, it does not replace all functionality of LOCK and must be implemented judiciously.

When the LOCKSAFE macro fails or is stopped manually, it can leave behind residue—semaphore and mutexes files littering the landscape. Thus, implementation of LOCKSAFE must include a fail-safe path—a program flow that executes in the event that a process or program must be terminated that had implemented LOCKSAFE. For example, if a program utilizing LOCKSAFE fails prematurely or is manually terminated, the likelihood exists that a semaphore file or mutex file exists that will need to be manually deleted. Until this manual deletion, other sessions of SAS attempting to gain access to the data set via LOCKSAFE might be unable to do so, believing that the associated data set remained locked. Thus, whenever LOCKSAFE is implemented, care must be exercised to ensure that these utility files—when created—are deleted before process or program termination and subsequent software recovery or resumption.

CONCLUSION

The LOCKSAFE solution replaces much of the functionality of the LOCK statement in that it facilitates secure, synchronized locking and unlocking of shared data sets. This enables inter-session communication between separate SAS instances via control tables that can be used to direct data-driven parallel processing paradigms. Because LOCKSAFE only offers security when all concurrent processes utilize the macro to gain access to a data set, LOCKSAFE must be implemented thoroughly within an infrastructure to ensure reliability. Notwithstanding this caveat, LOCKSAFE offers a comprehensive, stress-tested methodology to facilitate communication and functionality unavailable within Base SAS.

REFERENCES

[i] Hughes, Troy Martin. 2014. “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.” *Proceedings of the Twenty-Second Annual Western Users of SAS Software (WUSS) Conference.*

[ii] ISO/IEC/IEEE 24765:2010. Systems and software engineering—Vocabulary. Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers.

[iii] Chapman, Barbara, Jost, Gabriele, and van der Pas, Ruud. *Scientific and Engineering Computation: Using OpenMP : Portable Shared Memory Parallel Programming*. Cambridge, US: The MIT Press, 2007.

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

[v] SAS® 9.4 Statements: Reference, Fourth Edition. LOCK Statement. Retrieved from http://support.sas.com/documentation/cdl/en/lestmtsref/68024/HTML/default/viewer.htm#p0pc4mpj7xzxs4n1257fgz2tkuyb.htm

[vi] “The Alternate Side.” *Seinfeld*. 4 Dec. 1991. 3rd Season, 11th Episode. Fox. WNYW, New York City. Television.

[vii] ISO/IEC/IEEE 24765:2010. Systems and software engineering—Vocabulary.

[viii] Hughes, Troy Martin. *SAS Data Analytic Development: Dimensions of Software Quality*. New York, NY: John Wiley and Sons Publishing, 2016.

[ix] ISO/IEC/IEEE 21451-1:2010. Information technology—Smart transducer interface for sensors and actuators—Part 1: Network Capable Application Processor (NCAP) information model. Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers.

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. THE LOCKSAFE MACRO

```
%let mutlib=mutex;
libname &mutlib '/folders/myfolders/mutex'; * MUST CHANGE TO ACTUAL LOCATION;

%macro dtgx(dtg= /* datetime input for expansion to 3 decimal places */);
%do %while(%length(&dtg)<14);
    %if %length(&dtg)=10 %then %let dtg=&dtg..;
    %let dtg=&dtg.0;
    %end;
&dtg
%mend;

%macro sleep(sec= /* number of whole seconds to sleep */);
%local sleeping;
%if %upcase(%substr(&sysscp,1,3))=WIN %then %let sleeping=%sysfunc(sleep(&sec));
%else %do;
    data _null_;
       call sleep(&sec,1);
    run;
    %end;
%mend;

%macro slashdir();
%global slash;
%if %upcase(%substr(&sysscp,1,3))=WIN %then %let slash=\;
%else %let slash=/;
%mend;

%slashdir;


%macro locksafe(dsn= /* data set name in LIB.DSN format */,
    sec=1 /* seconds to wait (1 or greater) between lock reattempts */,
    max=60 /* maximum number of seconds to attempt lock until process timeout */,
    seccollect=1 /* seconds to wait (1 or greater) to collect all processes */,
    secoffset=0 /* seconds (to three decimal places) to add or subtract */,
    terminate=NO /* YES to unlock rather than lock a data set */);
%let syscc=0;
%global locksafeRC;
%let locksafeRC=LOCKSAFE FAILURE;
%local one sem dsid closed mutdsid lib tab dtgstart dtg filelist deleteme i j;
%let one=;
%let lib=%scan(&dsn,1);
%let tab=%scan(&dsn,2);
%let dtgstart=%dtgx(dtg=%sysevalf(%sysfunc(datetime(),15.3)+&secoffset));
%let dtg=&dtgstart;
%let mut=%sysfunc(pathname(&mutlib))&slash&lib._&tab..mutex;
filename mutfil "&mut";
* LOCK DATA SET;
%if %upcase(&terminate)=NO %then %do;
    %do %while(%sysevalf((&dtg-&dtgstart)<&max) and %length(&locksafeRC)>0);
       %let dtg=%dtgx(dtg=%sysevalf(%sysfunc(datetime(),15.3)+&secoffset));
```

```
        %if %sysfunc(fexist(mutfil))=0 %then %do;
           %if %length(&one)=0 %then %do;
              %let makesem=&lib._&tab._&dtg._&max._&sysjobid._.semaphore;
              filename semfil "%sysfunc(pathname(&mutlib))&slash&makesem";
              %let dsid=%sysfunc(fopen(semfil,u));
              %let closed=%sysfunc(fclose(&dsid));
              %end;
           %let one=&makesem;
           %sleep(sec=&seccollect);
           %let fsemRC=%sysfunc(filename(fsem,%sysfunc(pathname(&mutlib))));
           %let dsid=%sysfunc(dopen(&fsem));
           %if &dsid^=0 %then %do;
              %do i=1 %to %sysfunc(dnum(&dsid));
                 %let sem=%sysfunc(dread(&dsid,&i));
                 %if %length(&sem)>8 %then %do;
                        %if %substr(&sem, %length(&sem)-8)=semaphore %then %do;
                               %if %scan(&sem,1,_)=&lib and %scan(&sem,2,_)=&tab and
                                      &sem < &one and %sysevalf((%scan(&sem,3,_) +
                                      %scan(&sem,4,_) + 10) > &dtg) %then %let
                                      one=&sem;
                               %end;
                        %end;
                 %end;
              %let closed=%sysfunc(dclose(&dsid));
              %end;
           %if %scan(&one,5,_)=&sysjobid %then %do;
              %let dsid=0;
              %let i=1;
              %do %while(&dsid=0 and &i<1000);
                 %let dsid=%sysfunc(fopen(mutfil,u));
                 %let i=%eval(&i+1);
                 %end;
              %if &dsid=0 %then %let locksafeRC=LOCKSAFE failed writing mutex;
              %let closed=%sysfunc(fclose(&dsid));
              %let locksafeRC=;
              %end;
           %if %length(&locksafeRC)^=0 and &sec>1 %then %sleep(sec=&sec-1);
           %end;
        %else %sleep(sec=&sec);
        %end;
     %if &locksafeRC=LOCKSAFE FAILURE %then %let locksafeRC=LOCKSAFE timeout after
        &max seconds;
     %end;
* UNLOCK DATA SET;
%if %upcase(&terminate)=YES or (%length(&locksafeRC))>=16 %then %do;
     %if %upcase(&terminate)=YES or %upcase(%substr(&locksafeRC,1,16))=
           LOCKSAFE TIMEOUT %then %do;
        %if %upcase(&terminate)=YES %then %do;
           %let i=1;
           %let deleteme=1;
           %do %while(&deleteme^=0 and &i<1000);
              %let deleteme=%sysfunc(fdelete(mutfil));
              %let i=%eval(&i+1);
```

```
            %end;
         %if &deleteme^=0 %then %let locksafeRC=FDELETE of mutex failed;
         %end;
      * delete semaphores;
      %let fsemRC=%sysfunc(filename(fsem,%sysfunc(pathname(&mutlib))));
      %let dsid=%sysfunc(dopen(&fsem)); *this DOPEN could fail when concurrent;
      %if &dsid^=0 %then %do;
         %do i=1 %to %sysfunc(dnum(&dsid));
            %let sem=%sysfunc(dread(&dsid,&i));
            %if %length(&sem)>8 %then %do;
               %if %substr(&sem, %length(&sem)-8)=semaphore %then %do;
                     %if %scan(&sem,1,_)=&lib and %scan(&sem,2,_)=&tab and
                           %scan(&sem,5,_)=&sysjobid %then %do;
                           filename semfil
                                 "%sysfunc(pathname(&mutlib))&slash&sem";
                           %let j=1;
                           %let deleteme=1;
                           %do %while(&deleteme^=0 and &j<1000);
                                 %let deleteme=%sysfunc(fdelete(semfil));
                                 %let j=%eval(&j+1);
                                 %end;
                           %end;
                     %end;
               %end;
            %end;
            %let closed=%sysfunc(dclose(&dsid));
         %end;
      %let i=1;
      %do %while(%length(%scan(&filelist,&i,,S))>1);
         %let filname=%sysfunc(pathname(&mutlib))&slash%scan(&filelist,&i,,S);
         filename fils "&filname";
         %let deleteme=%sysfunc(fdelete(fils));
         %let i=%eval(&i+1);
         %end;
      %if %upcase(&terminate)=YES and &syscc=0 %then %let locksafeRC=;
      %end;
   %end;
%if &syscc>0 and %length(&locksafeRC)=0 %then %let locksafeRC=LOCKSAFE FAILURE;
%mend;
```