Paper BB-18-2015

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

Troy Martin Hughes

ABSTRACT

The LOCKITDOWN SAS® macro, introduced at WUSS in 2014, both detects and prevents data access collisions that occur when two or more SAS processes or users simultaneously attempt to access the same SAS data set. With the implementation of LOCKITDOWN, code reliability and robustness can be greatly improved through the elimination of this common source of process failure. And, as processes patiently wait for each other and play nicely without developer intervention or supervision, code autonomy and automation can be achieved. This autonomy, however, can hide inefficiencies that are created when processes continually vie for data set access and are forced to wait for each other repeatedly. This text introduces an expanded LOCKANDTRACK SAS macro that includes all functionality of LOCKITDOWN and which additionally records and monitors all successful (immediate and delayed) and unsuccessful file lock attempts through a unified control table. Metrics produced by LOCKANDTRACK can elucidate where process flow bottlenecks or inefficiencies exist, thus allowing developers to recalibrate file access stoplights for optimal throughput and performance.

INTRODUCTION

Data access collisions are a leading cause of process failure in Base SAS programming. Especially in production code that requires reliability and robustness, the existence and availability of a permanent data set should be demonstrated before access is attempted. Moreover, appropriate exception handling routines should reroute code if either the requested data set does not exist or if it is locked and in use by another process or user. The LOCKITDOWN macro, introduced at WUSS in 2014, detects shared and exclusive SAS file locks, waits if a file lock cannot be immediately achieved, and either successfully locks the data set for use or times out after a parameterized amount of time has passed[i]. When implemented within an exception handling framework, LOCKITDOWN acts as the stoplight that directs when processes can safely access shared data sets, thus facilitating robust, reliable, fault-tolerant programs that dynamically and defensively reroute program control to avoid failure.

As the reliability of SAS code increases, so too does its autonomy, as SAS practitioners are freed from their laborious log-checking responsibilities and can trust that code will execute correctly. In fact, the goal in many production environments is autonomous, scheduled code that runs without manual intervention. But even traffic stoplights that are functioning correctly must be calibrated to ensure they are efficiently handling the flow of traffic. For example, consider a stoplight that regulates traffic at a four-way intersection. If traffic flow is equivalent on the two intersecting roads, the green-to-red ratio would also be expected to be equal for each road, giving them equivalent right-of-way. However, if one road carries tremendously more traffic than the other, the stoplight should be calibrated commensurately to reflect greater green light time for that route so that its traffic can proceed. File access control devices (like LOCKITDOWN) similarly should be calibrated to ensure that they scale efficiently to meet the demands of increasing data throughput as well as the demands of increased users or processes requesting access to a specific data set. Thus, as a data environment matures and morphs, file access controls may need to be recalibrated.

While scalability of data volume, users, or dependent processes is a common reason to recalibrate a file access control, process priority also can influence this decision. A critical extract transform load (ETL) program might run infrequently or on few data but might also have the highest priority when it does execute. Developers would want to ensure that this critical process, despite its smaller throughput and frequency, takes precedence over other processes that might be requesting concurrent access to the same data sets. This same deference to precedence also is observed in vehicular traffic controls, as stoplights near fire stations often can be operated from the station to give priority to fire and rescue apparatuses. While only one or two vehicles may leave a station, these emergency signals ensure firefighters get on their way faster without impedance. Moreover, in many jurisdictions, traffic signal preemption devices allow emergency vehicles to signal stoplights throughout town to grant immediate right-of-way. Implementation of the LOCKANDTRACK macro can facilitate this same prioritization and efficiency, by detecting when critical processes are being delayed by less important processes, and by alerting developers to these inefficiencies through a control table that records all successful and unsuccessful file lock attempts.

The LOCKANDTRACK control table functions first and foremost as a control table, thus allowing SAS processes to access its observations and to dynamically alter program flow through data-driven processing. For example, if a SAS program TransformData requires exclusive access to a data set, the program can query the control table to determine if any process is using the data set and can respond accordingly. However, the control table secondly can operate as a repository of historic file lock metrics, thus allowing SAS practitioners to understand which processes are being delayed and which users or processes are causing those delays. And, by combining these two facets of the control table—data-driven processing and historical metrics—developers can implement intelligent, fuzzy logic program design. For example, an algorithm could automatically access the control table and assess that "on average, when program TransformData has data set CleanObs locked, the data set is locked for 90 minutes and, since the data set has only been locked for 5 minutes, I'm going to do another task rather than waiting around for 85 more minutes…" Thus, program flow could be rerouted immediately rather than tediously waiting over an hour for a separate process to complete and release the data set lock.

The LOCKANDTRACK macro, included in Appendix A, overcomes an inherent weakness in code that functions reliably all the time—autonomy that may obfuscate inefficiency. The LOCKITDOWN macro is required for LOCKANDTRACK usage, which can be located through the References section. For current users of LOCKITDOWN, the LOCKANDTRACK macro can be seamlessly integrated into existing exception handling program flow without change to structure or logic. And, for file access control tyros who are tired of SAS errors occurring from multiple processes or users attempting to simultaneously access the same data sets, implementation of LOCKANDTRACK will both prevent these errors as well as facilitate the monitoring and optimization of SAS process flows that share permanent SAS data sets.

LOCK INEFFICIENCIES

Although the intricacies of LOCKITDOWN are not discussed in this text, the LOCKITDOWN macro works by testing the availability (i.e., file existence and file lock status) of a SAS data set and subsequently deploying either a shared or exclusive lock, dependent upon the nature of the process. If a data set is unavailable initially (i.e., in use by another process or user), LOCKITDOWN enters a busy-waiting/spinlock cycle in which file access is continually tested until either the lock is achieved or the process times out. Return codes indicate this success or failure, thus LOCKITDOWN can easily be implemented into exception handling code that facilitates robust code. For more information about LOCKITDOWN or SAS locks in general, 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*.

Under ideal circumstances, a SAS process either is able to immediately access a requested data set or it waits an acceptable period of time (within the LOCKITDOWN busy-waiting cycle) before access is achieved. As a data infrastructure matures over time, however, additional users or dependent processes may vie for access to a permanent data set, causing it to be in use (i.e., locked) a greater proportion of the time. This can cause processes to slow as traffic jams form or can cause processes to fail as they exceed the parameterized maximum acceptable wait time. And, in a worst case scenario, a deadlock can result in which processes are waiting on each other and none can proceed.

Program runtime is often the primary observable metric of program performance, thus a program might be perceived as "slow" or "slower than in the past" while the cause of the inefficiency may not be fully understood. For example, over time, a program TransformData that once completed in 20 minutes might start taking 40 minutes, and then it might start failing entirely because it exceeds the maximum wait threshold (i.e., the MAX parameter) inside the LOCKITDOWN macro. In traditional SAS programming—in which a program runs continuously without interruption and then completes—typical causes of reduced efficiency may include poor programming habits, a larger volume of data, increased SAS server activity, inefficient hardware, or a slow SAS server or network connection. However, in SAS programs that implement file access controls such as LOCKITDOWN or LOCKANDTRACK, interruptions can occur whenever a data set is unavailable (i.e., locked) and the process must wait for file access. Thus, in programs that utilize file access controls, analysis of lock attempts should occur to ensure programs remain efficient and meet performance objectives, after which recalibration of these stoplights may be warranted.

In the above example in which the program TransformData uses the LOCKITDOWN macro, once the program started failing due to file access timeouts, developers hopefully would have observed this trend and recoded the LOCKITDOWN macro invocation to include a longer busy-waiting time threshold. This would allow the program, albeit significantly slowed, to continue to function without a timeout occurring. And, while the values of MAX can be increased ad infinitum to accommodate waiting hours or even days for a data set to become available, this can degrade performance to the point that business value is eliminated entirely. Thus, MAX typically should be set to the realistic amount of time that users are willing to wait for file access and, when exceeded causing a timeout, dependent processes must be eliminated or rescheduled, or the program itself must be refactored for increased efficiency.

LOCKANDTRACK EXAMPLE

Suppose that a company nightly updates an analytic SAS data set BigEnchilada that is subsequently used by scores of analysts throughout the day. Two years ago, this ETL program would complete correctly the vast majority of the time but, about every three weeks, someone or some process might have the BigEnchilada locked at night causing the update process to fail and causing substantial loss to business value the following day because the analytic data set had not been created. Thus, on these somber mornings, analysts either had no enchilada at all, or they had a day-old enchilada…and no one wants to eat (or use) a day-old enchilada. Although overly simplified, the ETL process creating the BigEnchilada can be represented with the following code.

```
data analysis.BigEnchilada;  /* requires exclusive lock to create this data set */
    set lilEnchilada;  /* requires shared lock to read from this data set */
run;
```

Along came the LOCKITDOWN macro, and suddenly the company's problems were solved. Now, if an analyst were working late and using the BigEnchilada or if another process had the data set locked, LOCKITDOWN would cause the ETL program to delay execution until the BigEnchilada was no longer in use. With this newfound robustness and reliability, developers stopped monitoring the ETL process because they could be confident that the BigEnchilada would be created—even if slightly delayed—and ready for analysts in the morning. The developers' upgraded code that implements LOCKITDOWN follows. Note that the &LOCKCLR statement is set to "LOCK TEST.BIGENCHILADA CLEAR;" in the LOCKITDOWN macro, and is required to release the explicit lock created by the LOCK statement inside the LOCKITDOWN macro.

```
%macro makin_enchiladas;
%LOCKITDOWN(lockfile=analysis.BigEnchilada, sec=1, max=21600, type=W,
    canbemissing=YES); /* 21600 sec = max wait time set for 6 hours */
%if %length(&lockerr)>0 %then %return; /* exit if data set locked after 6 hours */
data analysis.BigEnchilada;
    set lilEnchilada;
run;

&lockclr; /* releases explicit exclusive lock on BigEnchilada */
%mend;

%makin_enchiladas;
```

With the increased reliability and autonomy in the ETL process, other dependent processes could be reliably scheduled and the infrastructure continued to grow. And, to ensure that both ETL processes and analytic processes continued to play nicely, the LOCKITDOWN macro was implemented not only on the ETL process but also on analytic processes that requested read-only access to the BigEnchilada. However, with more dependent processes competing for read-only access to the BigEnchilada, developers began to notice that the BigEnchilada was being created later and later in the morning, and that occasionally analysts already had had their morning coffee and were ready to work but could not because the update had not completed—no longer because of process failure, but because of substantial delays in the ETL program. To be clear, the LOCKITDOWN macro was functioning perfectly as a traffic cop, directing file accesses and ensuring that no errors occurred. However, the high-priority ETL process was being forced to wait because required data sets were locked by lower priority programs. The developers moreover were somewhat stymied by the now sheer number of dependent processes requesting access to the BigEnchilada and weren't immediately certain which were causing the delay.

The implementation of LOCKANDTRACK solves the above conundrum, by illustrating in table format those processes that are forced to wait for data set access as well as those processes that timeout and cannot proceed because data sets are in use. The following code depicts the implementation of the LOCKANDTRACK macro on the above code, again using a simplified representation of the ETL process that creates the BigEnchilada.

```
%macro makin_enchiladas;
%LOCKANDTRACK(lockfile=analysis.BigEnchilada, sec=1, max=21600, type=W,
    canbemissing=YES); /* max wait time set for 6 hours */
%if %length(&locktrackerr)>0 %then %return; /* exit if data set locked after 6
hours */
data analysis.BigEnchilada;
    set lilEnchilada;
run;

%LOCKANDTRACK(lockfile=analysis.BigEnchilada, remove=Y);
%mend;

%makin_enchiladas;
```

Note that the revised LOCKANDTRACK implementation is virtually identical to the previous use of LOCKITDOWN. The invocation is identical, with no change to existing parameterized input, and only one added parameter to invoke the lock release. The LOCKTRACKERR error return code replaces the LOCKERR return code, which always should be assessed within an exception handling framework to ensure that if a lock cannot be established, the process or program does not continue to execute unwanted segments. The added parameter (REMOVE) clears the lock—whether shared or exclusive—using a second call to LOCKANDTRACK with the REMOVE=Y option. To clear the lock, only the data set name (LOCKFILE) and REMOVE parameters are required, as depicted in the above example.

Implementation of LOCKANDTRACK creates a control table that records all successful and unsuccessful lock attempts for all SAS data sets to which the macro is applied. In the above example, an observation might depict that at midnight, the MAKIN_ENCHILADAS macro attempted to lock the BigEnchilada data set, but that it could not gain access until 1:17 am. More importantly, however, it also depicts that at midnight, the BigEnchilada was locked by a program called AnalyzeThis and had been exclusively locked since 11:40 pm. Analysis of this single observation can help developers hone their infrastructure by identifying process flow impediments. One suggestion would be to reschedule the AnalyzeThis program to a later time (earlier in the morning) after the BigEnchilada had definitely been created. Another suggestion would be to have AnalyzeThis—an analytic program, that should in no way be modifying the BigEnchilada—use a shared rather than exclusive file lock when LOCKANDTRACK is invoked. And, when data from the control table were cumulatively analyzed over time, trends might emerge that could depict where other common file access conflicts, inefficiencies, or process bottlenecks exist.

Thus, some process inefficiencies can be solved simply by modifying LOCKITDOWN or LOCKANDTRACK parameters, for example, by decreasing the SEC parameter so that file lock testing occurs more frequently and is thus prioritized over other processes. In other cases, by increasing the MAX parameter, a process that initially times out and fails to establish a data set lock can function correctly again if the maximum wait time is increased. And, many data access conflicts can be solved by scheduling processes more appropriately so they do not overlap or overlap less substantially. Each of these remedies is rather straightforward and equates to stoplight recalibration; the same intersection will exist, but it will operate more efficiently due to feedback from LOCKANDTRACK metrics. In other cases, however, the solution is less facile and may involve substantial redesign or a total overhaul of the existing intersection. For example, perhaps the infrastructure is bottlenecked and has outgrown the four-way intersection and an overpass is warranted that will permit continuous traffic flow from all directions. In a data environment, this might require the creation of duplicate data sets so that more processes can access them simultaneously, or system solutions such as the implementation of SAS/SHARE, which can facilitate concurrent data set access. Regardless of which methodology is selected to improve throughput and efficiency, developers should match their solution and level of effort with specific performance requirements to ensure they are neither delivering a gold-plated solution that is overkill nor a bandage that inadequately attempts to buttress a flailing infrastructure.

LOCKANDTRACK SETUP AND INVOCATION

The LOCKANDTRACK macro requires installation of the LOCKITDOWN macro, referenced in the References section. While a best practice is to implement both LOCKITDOWN and LOCKANDTRACK through the SAS Autocall library, at the very least, an %INCLUDE statement must reference the location of the LOCKITDOWN macro. The second requirement is the creation of the library LOCKNTRK in which the LOCKANDTRACK control table will be located. The logical location of the library (i.e., Windows or UNIX folder path) should exist before invocation, but the library itself can be assigned during the initial LOCKANDTRACK execution with the LIBNAME statement. Prior to use of the macro, two lines of code must be modified locally by developers, each line demarcated with /* MUST BE MODIFIED */.

The LOCKANDTRACK macro is invoked using parameters identical to the LOCKITDOWN macro, with the exception that the optional REMOVE parameter specifies that a lock is being released rather than established.

```
%macro LOCKANDTRACK(lockfile= /* data set in LIBRARY.DATASET or DATASET
           format */,
    sec=5 /* interval in seconds after which data set is retested*/,
    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 */,
    remove= /* (Y) or (YES) to release a current lock */);
```

- LOCKFILE – Data sets typically only need to be locked when they reside in permanent SAS libraries that are accessible to other users and to concurrent SAS sessions. Notwithstanding, LOCKANDTRACK can be used on data sets in the WORK library by omitting the library token in the LOCKFILE parameter but, in practice, this should never be done.

- SEC – If a lock cannot be achieved, the SAS session will sleep a number of seconds before reattempting access. Thus, this parameter also acts to prioritize sessions, some of which simultaneously may be waiting for the same locked data set. For example, a program with a 1 second sleep interval will be approximately ten times more likely to achieve a lock than a program simultaneously 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 their SEC parameter.
- MAX – After an unacceptable maximum delay, the *event*—a locked file—becomes an *exception* and the macro times out, returning a corresponding error return code. 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 to release locks in sequence on the same data set.
- TYPE – The appropriate lock type—either exclusive (i.e., read-write) or shared (i.e., read-only)—must be chosen contextually based on the required action. 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 required read-write access. Or, conversely, requesting a read-write lock when only a read-only lock is required unnecessarily locks out other processes from the data set.
- CANBEMISSING – This parameter 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 thereafter, in which case it would not exist during the initial iteration. If the data set conversely can never be missing, LOCKANDTRACK first determines file existence and, if the data set does not exist, halts and produces a return code indicating this error.
- REMOVE – When this optional parameter is included, it indicates that a lock is being released rather than established. Moreover, when REMOVE is included, only the LOCKFILE parameter must also be included in the macro invocation.

When LOCKANDTRACK is invoked, error handling routines first ensure that the data set exists (if required), after which the current lock status is tested and, if the required lock is available, the lock is obtained. If the lock is unavailable, the macro waits and reattempts, maintaining this busy-waiting cycle until either the lock is achieved or the process times out. LOCKANDTRACK differs from LOCKITDOWN in that after this final resolution—a lock obtained immediately, a lock obtained after a delay, or a failure due to time out—the resolution is recorded as an observation in the control table LOCKNTRK.CONTROL. This allows all other users and processes to know immediately which process has secured the data set lock as well as when the lock was established.

After a lock is achieved and data set access has occurred and concluded, the lock must be manually released. Thus, in all cases, and as demonstrated in the example above, the LOCKANDTRACK macro must be invoked a second time with the REMOVE parameter included. Without this step, failure to release the lock—especially if it is an exclusive lock—can result in subsequent delayed or failed processes for the program that is running or for other concurrent users or processes.

LOCKANDTRACK CONTROL TABLE

The control table (LOCKNTRK.CONTROL) represents a historical repository of all data set locks that are created as well as those that are attempted yet denied. It contains invaluable metrics that, whether used singly or in aggregate, can provide developers with better situational awareness of their data infrastructure and which can be used to eliminate file access collisions and to reduce process bottlenecks. To be effective, however, if LOCKANDTRACK is implemented on a permanent data set, it must be implemented across all programs and processes that access or utilize that data set. Moreover, users no longer can simply double-click on the data set—even briefly—to view it, but instead must insist on using code containing LOCKANDTRACK for all data set access.

Thus, while LOCKANDTRACK logic can be implemented sparingly on as few or as many permanent data sets as are warranted, once a data set is registered in the control table, LOCKANDTRACK should always be used anytime that data set is referenced in any way. Otherwise, the value of the macro is lost. For example, if the data set BigEnchilada is always accessed via LOCKANDTRACK, every file access attempt as well as the duration of all successful file locks are captured in the control table. However, if a SAS procedure or DATA step accesses the BigEnchilada without first invoking LOCKANDTRACK, this access (and its corresponding file lock) will not be recorded in the control table. And, if a subsequent process uses LOCKANDTRACK and attempts to concurrently access the BigEnchilada while the data set is still in use, the macro LOCKANDTRACK will 1) recognize that the data set is locked (thus preventing a file access collision and error) but 2) will be unable to inform the developer which user or process is using the BigEnchilada. Instead, in this example, the control table fields LOCKEDBY_PROGRAM and LOCKEDBY_PROCESS simply will indicate "UNKNOWN".

The LOCKANDTRACK control table contains the following fields:

- LIB – the SAS library in which the data set (that is being locked) resides.
- FNAME – the SAS data set name.
- DSID – the data set ID, which is used to identify data sets opened with a shared lock, and which is used subsequently to close these data sets using the %SYSFUNC(CLOSE(&DSID)) function.
- PROGRAM – dynamically generated name of the SAS program requesting the file lock. The value is dynamically generated in Windows using the %SYSGET(SAS_EXECFILENAME) statement and in UNIX using the &_SASPROGRAMFILE automatic macro variable.
- PROCESS – manually generated name of the SAS process requesting the file lock (process must be manually supplied by developers and passed to the LOCKANDTRACK macro using the global macro variable &LOCKTRACKPROCESS.) If no value is supplied by developers, this field will be missing.
- SEC – interval in whole-number seconds that the macro waits between file lock attempts, also used to prioritize one process over others, with higher priority processes having lower second values.
- MAX – maximum seconds that the macro churns before timing out and producing a failure return code.
- TYPE – the type of lock that was requested (either R for read-only or W for read-write.)
- DTG_REQUEST – date and time of the initial file lock request.
- DTG_TIMEOUT – date and time of the file lock timeout (if applicable and the request timed out, this value should approximate the value MAX added to the DTG_REQUEST.)
- DTG_LOCKSTART – if a lock was successful, the date and time that the lock started. This value is missing if the lock times out.
- DTG_LOCKSTOP – if a lock was successful, the date and time that the lock ended (supplied by the second invocation of LOCKANDTRACK with the REMOVE=Y parameter.) This value is missing if the lock times out.
- LOCKEDBY_PROGRAM – If the file lock was not immediately available or if LOCKANDTRACK timed out before it could gain access to a data set, the program that maintained the original lock is recorded here. The value is dynamically generated in Windows using the %SYSGET(SAS_EXECFILENAME) statement and in UNIX using the &_SASPROGRAMFILE automatic macro variable. However, if LOCKANDTRACK is not used to access a data set that is locked by a program, then the control table entry will only indicate "UNKNOWN" for this value.
- LOCKEDBY_PROCESS – If the file lock was not immediately available or if LOCKANDTRACK timed out before it could gain access to a data set, the process that maintained the original lock is recorded here (process must be manually supplied by developers and passed to the LOCKANDTRACK macro using the global macro variable &LOCKTRACKPROCESS.) However, if LOCKANDTRACK is not used to access a data set that is locked by a program, then the control table entry will only indicate "UNKNOWN" for this value. If no value is supplied by developers, this field will be missing.
- LOCKEDBY_START – If the file lock was not immediately available or if LOCKANDTRACK timed out before it could gain access to a data set, this value will indicate the date and time that the initial process gained the file lock.

The control table will continue to grow cumulatively as each data set is locked and respectively unlocked and, because a single process might need to lock a data set multiple times and might execute on an hourly or even more frequent basis, the control table can quickly grow unwieldy. A decision will need to be made regarding how many observations to maintain, which should be based on the intent of the control table. The control table can be used solely as a lookup table to help manage and track current data set locks, thus facilitating data-driven processing. In this case, a SAS script that deletes all observations older than 24 or 48 hours might run each night to ensure the control table remains fresh. And, developers might be able to identify inefficient or conflicting processes from single observations, obviating the need to maintain large registries of historic file lock data. But, to reiterate, the most powerful aspect of LOCKANDTRACK its ability to operate as a control table that is driven by both historical norms (of file lock metrics) as well as real-time values depicting current lock statuses. And, to fully exercise this potential, a month or months' worth of lock data should be maintained.

CONCLUSION

The SAS macro LOCKANDTRACK includes all functionality of the LOCKITDOWN macro, thus preventing vexing data set access collisions caused by competing, concurrent processes or users. Moreover, LOCKANDTRACK expands this functionality by tracking all successful and unsuccessful file lock requests, allowing developers to monitor and optimize their code and the scheduling and automation thereof. Like all quality control techniques that are implemented to increase software reliability and robustness, implementation of LOCKANDTRACK inherently carries a tradeoff—quality is increased but at the price of code volume and complexity. Thus, tools such as LOCKITDOWN and LOCKANDTRACK should be implemented only based on a software performance requirement, which typically will be the need for robust, reliable, fault-tolerant SAS software. But, with its implementation, LOCKANDTRACK can facilitate code that is not only reliable and autonomous but also efficient.

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*. Western Users of SAS Software (WUSS).

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 – LOCKANDTRACK MACRO CODE

```
%macro LOCKANDTRACK(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 requested */,
    canbemissing=N /* either (Y) or (N), indicating if data set can not exist */,
    remove=N /* only Y or YES when removing a lock after file use */);
%if %sysfunc(libref(lockntrk))^=0 %then %do;
    libname lockntrk '/folders/myfolders/lockandtrack'; /* MUST BE MODIFIED */
    %end;
%include '/folders/myfolders/lockitdown.sas'; /* MUST BE MODIFIED */
%local i;
%local lib;
%local tab;
%local starttime;
%local max;
%local loc;
%global dsid;
%global locktrackerr;
%global locktrackclr;
%let locktrackerr=;
%let locktrackclr=;
%let dsid=;
* initialize the control table if necessary;
%if %sysfunc(exist(lockntrk.control))^=1 %then %do;
    data lockntrk.control;
          length lib $12 fname $32 dsid 8 program $32 process $32 sec 8 max 8 type
$2
                dtg_request 8 dtg_timeout 8 dtg_lockstart 8 dtg_lockstop 8
                lockedby_program $32 lockedby_process $32 lockedby_start 8;
          format dtg_request dtg_timeout dtg_lockstart dtg_lockstop lockedby_start
datetime17.;
    run;
    %end;
* standardize input;
%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 not %symexist(locktrackprocess) %then %let locktrackprocess=;
%if &SYSSCP=WIN %then %do;
    %let sys=WIN;
    %let locktrackprogram=%sysget(SAS_EXECFILENAME);
    %end;
%else %do;
    %let sys=UNIX;
    %let locktrackprogram=&_SASPROGRAMFILE;
    %end;
%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 locktrackerr=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 locktrackerr=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 locktrackerr=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 locktrackerr=LOCKITDOWN failed because value for TYPE must be either R or
W;
    %goto err;
    %end;
%if &canbemissing^=Y and &canbemissing^=N %then %do;
    %let locktrackerr=LOCKITDOWN failed because CANBEMISSING must be Y or N;
    %goto err;
    %end;
* lock removal;
%if %upcase(&remove)=Y or %upcase(&remove)=YES %then %do;
    %lockitdown(lockfile=lockntrk.control, sec=1, max=300, type=W,
canbemissing=NO);
    data lockntrk.control;
           set lockntrk.control;
           if _n_=1 then call symput("found",0);
           if lib="%upcase(&lib)" and fname="%upcase(&tab)" and not
missing(dtg_lockstart) and
                          missing (dtg_lockstop) then do;
                  dtg_lockstop=%sysfunc(datetime());
                  call symput("dsid",dsid);
                  call symput("type",type);
                  call symput("found",1);
                  end;
    run;
    lock lockntrk.control clear;
    %if &found=1 %then %do;
           %if &type=W %then %do;
                  lock &lib..&tab clear;
                  %end;
           %else %do;
                  %let x=%sysfunc(close(&dsid));
                  %end;
           %end;
    %goto noerr;
    %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 locktrackclr=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; /* timeout before lock achieved */
    %let locktrackerr=LOCKANDTRACK failed after %sysevalf(%sysfunc(datetime())-
&starttime)
                  seconds to gain %sysfunc(ifc(&type=W,an exclusive,a shared)) lock
                  on &lockfile;
    %lockitdown(lockfile=lockntrk.control, sec=1, max=300, type=W,
canbemissing=NO);
    data lockntrk.control (drop=found temp_lockedby_program temp_lockedby_process
                  temp_lockedby_start);
           set lockntrk.control end=eof;
           output;
           if _n_=1 then found=0;
           length temp_lockedby_program $32 temp_lockedby_process $32
temp_lockedby_start 8;
           format temp_lockedby_start datetime17.;
           if lib="%upcase(&lib)" and fname="%upcase(&tab)" and not
missing(dtg_lockstart) and
                         missing(dtg_lockstop) then do;
                  found=1;
                  temp_lockedby_program=program;
                  temp_lockedby_process=process;
                  temp_lockedby_start=dtg_lockstart;
                  put temp_lockedby_program;
                  end;
           retain temp_lockedby_program temp_lockedby_process temp_lockedby_start
found;
           if eof then do;
                  lib="%upcase(&lib)";
                  fname="%upcase(&tab)";
                  dsid=&dsid;
                  program="%upcase(&locktrackprogram)";
                  process="%upcase(&locktrackprocess)";
                  sec=&sec;
                  max=&max;
                  type="%upcase(&type)";
                  dtg_request=&starttime;
                  dtg_timeout=%sysfunc(datetime());
                  dtg_lockstart=.;
                  dtg_lockstop=.;
                  if found=1 then do;
                         lockedby_program=temp_lockedby_program;
                         lockedby_process=temp_lockedby_process;
                         lockedby_start=temp_lockedby_start;
                         end;
                  else do;
                         lockedby_program="UNKNOWN";
                         lockedby_process="UNKNOWN";
                         lockedby_start=.;
                         end;
                  output;
                  end;
    run;
    lock lockntrk.control clear;
    %end;
%else %do; /* lock achieved */
    data control_temp;
           length lib $12 fname $32 dsid 8 program $32 process $32 sec 8 max 8 type
$2
                  dtg_request 8 dtg_timeout 8 dtg_lockstart 8 dtg_lockstop 8
                  lockedby_program $32 lockedby_process $32 lockedby_start 8;
           format dtg_request dtg_timeout dtg_lockstart dtg_lockstop lockedby_start
datetime17.;
           lib="%upcase(&lib)";
           fname="%upcase(&tab)";
           dsid=&dsid;
           program="%upcase(&locktrackprogram)";
           process="%upcase(&locktrackprocess)";
           sec=&sec;
           max=&max;
           type="%upcase(&type)";
           dtg_request=&starttime;
           dtg_timeout=.;
           dtg_lockstart=%sysfunc(datetime());
           dtg_lockstop=.;
           lockedby_program="";
           lockedby_process="";
           lockedby_start=.;
    run;
    %lockitdown(lockfile=lockntrk.control, sec=1, max=300, type=W,
canbemissing=NO);
    proc append base=lockntrk.control data=control_temp;
           run;
    lock lockntrk.control clear;
    %end;
%err: %put &locktrackerr;
%noerr:;
%mend;
```