

Paper AD-209

A Waze App for Base SAS®: Automatically Routing Around Locked Data Sets, Bottleneck Processes, and Other Traffic Congestion on the Data Superhighway

Troy Martin Hughes

ABSTRACT

The Waze application, purchased by Google in 2013, alerts millions of users about traffic congestion, collisions, construction, and other complexities of the road that can stymie motorists' attempts to get from A to B. From jackknifed rigs to jackalope carcasses, roads can be gnarled by gridlock or littered with obstacles that impede traffic flow and efficiency. Waze algorithms automatically reroute users to more efficient routes based on user-reported events as well as historical norms that demonstrate typical road conditions. Extract-transform-load (ETL) infrastructures often represent serialized process flows that can mimic highways, and which can become similarly snarled by locked data sets, slow processes, and other factors that introduce inefficiency. The LOCKITDOWN SAS® macro, introduced at WUSS in 2014, 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. Moreover, the LOCKANDTRACK macro, introduced at WUSS in 2015, provides real-time tracking of and historical performance metrics for locked data sets through a unified control table, enabling developers to hone processes to optimize efficiency and data throughput. This text demonstrates the implementation of LOCKSMART and its lock performance metrics to create data-driven, fuzzy logic algorithms that preemptively reroute program flow around inaccessible data sets. Thus, rather than needlessly waiting for a data set to become available or a process to complete, the software actually predicts the wait time based on historical norms, performs other (independent) functions, and returns to the original process when it becomes available.

INTRODUCTION

One of the most common sources of functional failure within production SAS software is the threat of multiple users or processes attempting simultaneously to access the same data set. If a data set is being created or modified, the SAS session maintains an exclusive lock during the process which prevents any other user or process from modifying or even viewing the data set. Thus, anytime an exclusive lock is held on a data set within a shared library accessible by other users or processes, this lock poses a threat to other concurrent attempts to access the data set because they will fail. The LOCKITDOWN macro eliminates this risk by testing data set locks before data set use and is described extensively in a separate text by the author: *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*.¹ The LOCKSMART solution demonstrated in this text requires the LOCKITDOWN macro.

With implementation of LOCKITDOWN within an exception handling framework, production SAS software can be automated, scheduled, and reliably executed without fear of file access collisions. Because of this newfound reliability and robustness, SAS practitioners are freed to pursue more interesting tasks than babysitting tenuous software or log parsing in search of runtime errors. However, as permanent data sets become more popular (because they can now be reliably accessed without fear of failure), SAS processes may begin to experience increasingly longer wait times to access data sets via the LOCKITDOWN macro. For example, if a critical data set is depended upon by 10 different processes, each of those dependencies will be competing for shared or exclusive file locks which can slow software performance. Thus, data sources that are extremely reliable can become victims of their own success as additional users learn of their reliability and clamor for access.

To facilitate identification of SAS processes that may be unnecessarily using or monopolizing data sets for extended periods of time, the LOCKANDTRACK macro was introduced which creates a historical control table that records all attempted and successful SAS file locks. Through the analysis of *successful* file locks, SAS practitioners can quantify how long specific SAS processes require control over specific SAS data sets, thus demonstrating processes that might benefit from performance improvement. And, through the analysis of *failed* file locks, SAS practitioners can determine which processes are blocking other processes from accessing necessary permanent data sets. Armed with

this information, SAS practitioners are better equipped to schedule SAS batch jobs so they don't conflict with data access requests from other processes. Moreover, where programs are identified that cause substantially long file locks (thus delaying other processes), those programs can be specifically targeted to be made faster or rescheduled.

The LOCKANDTRACK macro is introduced in a separate text by the author: *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*ⁱⁱ. The LOCKSMART solution demonstrated in this text replaces the LOCKANDTRACK macro but the separate text is essential to understanding the mechanics of the LOCKSMART macro. This text continues that discussion to demonstrate how—with sufficient historical lock performance data for a specific process on a specific data set—those statistics can be programmatically accessed at the point a file lock is requested not only to alert a process whether a data set is locked (and inaccessible), but also to notify the process of the predicted wait time required for access. For example, when a process executes and attempts to access a data set, LOCKSTART will query the control table, note that the data set has been locked for 20 minutes and (on average) will continue to be locked for another 40 minutes, and autonomously decide to perform some other function in lieu of waiting.

The benefits of LOCKSTART are tremendous. Data analytic infrastructures often resemble highways with processes attempting to transport data from one form or location to another as quickly as possible. Progress can be impeded, however, where consistent bottlenecks exist, such as multiple processes competing for access to the same data set or data sets that are otherwise locked for unreasonable periods of time. Just as the Waze application (described in the next section) relies on real-time conditions and historical highway metadata to predict the severity and duration of traffic in specific locations, SAS software can also utilize historical norms of file lock trends to programmatically enable processes to route around data sets that are locked and which are expected to be locked for a considerable amount of time.

THE WAZE APPLICATION

The Waze mobile application provides real-time traffic-related reporting to users to enable them to route around accidents, construction, traffic jams, and other sources of congestion that are reported by users. Other potential hazards such as potholes, broken-down cars, parked emergency vehicles, and other objects in or near roadways can also be reported with only the swipe of a finger. Figure 1 demonstrates the Waze application map showing gnarled traffic while crossing the Grapevine on the Five in Lebec, California, with police activity, at least two accidents, and several miles of traffic congestion.

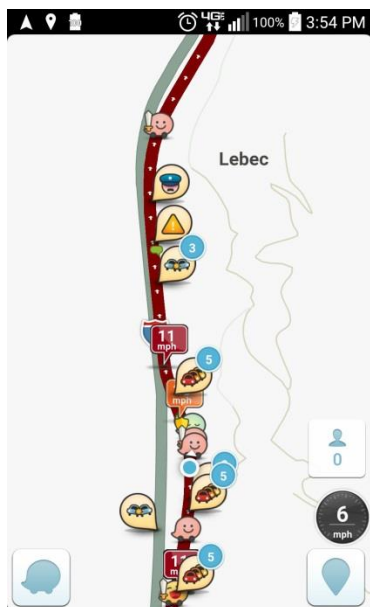


Figure 1. The Waze Application Crossing the Grapevine

Although Waze provides real-time data for a driver's current direction of travel, it is most useful when a destination is inputted so that Waze can route drivers using the fastest method, redirecting them if necessary around obstacles. For example, Google Maps uses Waze data to determine preferred routes between two locations based on shifting traffic, thus enabling mobile users to receive an audible and visual notification stating "A faster route is now available. Click to accept that route." In addition to relying on manually-inputted user reports that describe traffic and road conditions, Waze also automatically collects metadata such as user location and driving speed to further validate flow of traffic. When traffic is analyzed with respect to traffic incidents that have occurred, historical traffic norms can be created that are specific to time of day and day of week and which even account for seasonal variations such as additional school traffic during the school year. These historical metadata enable Google not only to provide real-time traffic updates but also to predict future traffic for a specific day, time, location, and direction. And, if Waze were not already awesome enough, Waze users are rewarded with "road candy" for describing traffic conditions, validating the reports of other users, or achieving other milestones, thus facilitating increased data quality and integrity.

DATA ANALYTIC SCENARIO

Consider the scenario in which an ETL process generates a persistent data set PERM.Critical a few times per week on average. However, smaller transactional updates are made to the data set every four hours, thus two SAS macro processes—CREATE and UPDATE—require exclusive locks on the data set. Other analytic processes also rely on PERM.Critical but, because they do not modify the data set, require only shared file locks. However, these analytic processes—represented as the macros FREQ and MEANS—cannot execute while an exclusive lock is maintained by CREATE or UPDATE. Thus, the business problem that exists is that multiple analytic processes may be stuck waiting for an exclusive lock to be released on the same data set. If the analytic processes have to wait only a couple minutes for CREATE or UPDATE to complete, this is rather efficient. However, if the CREATE macro is executing (which takes approximately two hours) and still has substantial time remaining, it would be more efficient to shift program flow to unrelated functions rather than to linger in a busy-waiting spinlock for over an hour.

In this scenario, to determine what process has locked the PERM.Critical data set, the first requirement is that all SAS processes that reference the PERM.Critical data set must utilize the LOCKANDTRACK (or LOCKSMART) macro within an exception handling framework. LOCKANDTRACK gains its integrity from the assurance that all lock attempts—successful and failed—are recorded in the LOCKNTRK.Control control table. If the CREATE macro fails to register that it successfully locks the PERM.Critical data set, other processes (such as the two analytic processes) will not be aware of the average duration of the lock when they attempt to open PERM.Critical. Thus, the following CREATE macro definition invokes the LOCKANDTRACK macro once to lock the data set and a second time (with the REMOVE=YES parameter) to terminate the file lock:

```
* simulates module that less frequently fully updates data;
%macro create();
%let locktrackprocess=&sysmacroname;
%LOCKANDTRACK(lockfile=perm.critical, sec=1, max=1800, type=W,
  canbemissing=NO);
data perm.critical (drop=i);
  length num1 8 char1 $8;
  do i=1 to 100000000;
    num1=round(rand('uniform')*100);
    char1=byte(round(rand('uniform')*25)+65); * letters A to Z;
    output;
  end;
run;
%LOCKANDTRACK(lockfile=perm.critical, remove=YES);
%mend;
```

A critical process like the CREATE or UPDATE macros would most likely need to wait for the PERM.Critical data set to be unlocked if it attempted access and the data set were locked. For example, if an analytic process like MEANS had a shared lock on PERM.Critical, neither the CREATE nor UPDATE macros would be able to execute because each requires an exclusive lock. However, because of their respective criticality to the overall data infrastructure, both

CREATE and UPDATE would need to wait as long as possible to obtain the exclusive lock. For this reason, they would need to utilize the LOCKANDTRACK macro while specifying a high MAX parameter, which describes the amount of time the process is willing to wait in a busy-waiting cycle for a data set to be unlocked.

A less significant macro module like MEANS or FREQ, however, could benefit by not waiting for PERM.Critical if the data set were locked and were expected to be locked for some extended period of time. The LOCKSMART macro, demonstrated in Appendix A, is invoked (in lieu of LOCKANDTRACK) and thus additionally queries historical file lock statistics to drive program flow dynamically. In this example, if LOCKSMART cannot gain a shared (i.e., read-only) file lock on the PERM.Critical data set within 600 seconds (while checking the lock status once per second), this exception is noted in the &LOCKTRACKERR return code (which borrows the return code name from the LOCKANDTRACK macro), and program flow skips and does not execute the FREQ module:

```
%locksmart(dsn=perm.critical, sec=1 ,max=600, type=R, canbemissing=NO)
%if %length(&lockcheckRC)=0 %then %do;
    %freq();
    %locksmart(lockfile=perm.critical, remove=YES);
%end;
* do other stuff if lock not available;
```

In this example, when LOCKSMART is invoked, it queries the LOCKNTRK.Control control table, determines that the CREATE process currently has the data set PERM.Critical locked, determines that the process has had the data set locked for 20 minutes, and determines that on average when the CREATE process locks this data set it is locked for 60 minutes. Thus, the exclusive lock on the data set will be expected to last for another 40 minutes, while the MAX parameter specified by the LOCKSMART macro invocation is only ten minutes. Because the anticipated wait time (40 minutes) exceeds the MAX parameter (10 minutes), the LOCKSMART macro will exit and the &LOCKTRACKERR return code will indicate that the attempt to access the data set was aborted (for exceeding the maximum wait time). Although the FREQ module will not execute, this returns program control to the parent process calling LOCKSMART and enables it to perform subsequent, unrelated tasks rather than needlessly waiting for the data set lock to be released.

However, if the CREATE process instead had already been running for 55 minutes and thus had only five minutes remaining (on average, based on historical norms in the control table), the LOCKSMART macro would instead enter a busy-waiting cycle in which it tested the availability of the data set PERM.Critical every second for ten minutes or until it gained access to the data set. Thus, rather than unintelligently waiting to achieve a file lock (or until the MAX parameter threshold is reached), LOCKSMART has foresight into the expected lock duration and through fuzzy logic algorithms can more efficiently enable parent processes to reroute program flow. In actual implementation, more complex business logic might have instructed that if the FREQ module had not completed, it would have been attempted later in the program flow.

LOCKSMART MACRO

The invocation of the LOCKSMART macro and its return code (&LOCKTRACKERR) are identical to LOCKANDTRACK to facilitate extensibility from one method to the other. An initial invocation is required to record the lock attempt in the LOCKNTRK.Control control table while a subsequent invocation (with the REMOVE=YES parameter) is required to record the ultimate lock success or failure:

```
%macro locksmart(lockfile= /* data set in LIB.DSN or DSN format */,
    sec=5 /* interval in seconds for retesting file lock */,
    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), if DSN can not exist */,
    remove= /* (Y) or (YES) to release a current lock */);
```

- **LOCKFILE** – Data sets typically need to be locked only when they reside in permanent SAS libraries that are accessible to other users and to concurrent SAS sessions. Notwithstanding, LOCKSMART can be used on data

sets in the WORK library by omitting the library token in the LOCKFILE parameter but, in practice, LOCKSMART has little benefit when used in the WORK library.

- **SEC** – If a lock cannot be achieved, the SAS session enters a busy-waiting cycle in which it sleeps 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 one second sleep interval will be approximately ten times more likely to achieve a lock than a program simultaneously attempting access but using a ten second sleep interval. In this manner, a more critical process can be prioritized over less significant ones by reducing its SEC parameter.
- **MAX** – After an unacceptable maximum delay, the *event*—a locked file—becomes an *exception* and the macro times out, returning a corresponding return code. When transactional or standardized data sets are being processed, process time estimation is typically highly correlated with file size and thus can be estimated with some degree of certainty. If multiple processes are expected 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 successively to obtain and release locks 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 process obtaining only read-only access and producing a runtime error if it actually required read-write access. Or, conversely, requesting an exclusive lock when only a shared lock is required unnecessarily blocks other processes from simultaneously using 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, then CANBEMISSING should be set to NO so that LOCKSMART will test file existence, and if the data set does not exist, exit and produce a return code indicating this exception.
- **REMOVE** – When this optional parameter is included, it indicates that a lock is being released rather than established. When REMOVE is included, only the LOCKFILE parameter also must be included in the macro invocation.

LOCKSMART SETUP

As mentioned, implementation of the LOCKSMART solution requires the LOCKITDOWN macro, which can be obtained online. The LOCKITDOWN macro can be downloaded, saved as a SAS program (without modification), and referenced via the %INCLUDE statement. In this example, the macro has been saved to the SAS program C:\perm\lockitdown.sas so the following %INCLUDE statement is required before invocation of LOCKITDOWN or LOCKSMART:

```
%include 'c:\perm\lockitdown.sas';
```

Use of LOCKSMART may follow prior implementation of LOCKANDTRACK, in which case historical performance metrics should already exist in the LOCKNTRK.Control control table. For example, the LOCKANDTRACK macro might be used to monitor a specific data set for a month, after which performance data within the control table could be used to drive the LOCKSMART macro. In other cases, no performance data will exist and the LOCKSMART macro will be generating the performance data itself. In this example, the LOCKSMART macro can be downloaded from Appendix A and saved to the SAS program C:\perm\locksmart.sas. However, before it can be executed, two lines of code (distinguished with “MUST BE MODIFIED”) will need to be modified to reflect the host environment:

```
%if %sysfunc(libref(lockntrk)) ^=0 %then %do;
    libname lockntrk 'c:\perm\lockntrk'; * MUST BE MODIFIED;
%end;
%include 'c:\perm\lockitdown.sas'; * MUST BE MODIFIED;
```

In this scenario, a sample control table can be generated with the following code, which also must incorporate the CREATE macro. Note that the code must be saved into a SAS program (such as C:\perm\locktest.sas) to function because LOCKSMART cannot be invoked from an unnamed program. Also note that whereas the previous CREATE macro utilized LOCKANDTRACK, the revised CREATE macro now invokes LOCKSMART:

```
* THIS PROGRAM MUST BE SAVED AS A NAMED FILE SUCH AS C:\PERM\LOCKTEST.SAS;
libname perm 'c:\perm';
%include 'c:\perm\locksmart.sas';

* simulates module that less frequently fully updates data;
%macro create();
%let locktrackprocess=&sysmacroname;
%locksmart(lockfile=perm.critical, sec=1, max=1800, type=W,
  canbemissing=NO);
data perm.critical (drop=i);
  length num1 8 char1 $8;
  do i=1 to 10000000;
    num1=round(rand('uniform')*100);
    char1=byte(round(rand('uniform')*25)+65); * letters A to Z;
    output;
  end;
run;
%locksmart(lockfile=perm.critical, remove=Y);
%mend;

* simulates module that more frequently performs a partial update of data;
%macro update();
%let locktrackprocess=&sysmacroname;
%locksmart(lockfile=perm.critical, sec=1, max=1800, type=W,
  canbemissing=NO);
data perm.other;
  set perm.critical;
run;
%locksmart(lockfile=perm.critical, remove=Y);
%mend;

%macro iterate;
%global locktrackprocess;
%do doit=1 %to 10;
  %create;
  %update;
%end;
%mend;

%iterate;
```

LOCKSMART must be executed only from within a named program because the program name is programmatically assessed and saved within the LOCKNTRK.Control control table. The &LOCKTRACKPROCESS macro variable is set equal to the name of the macro that is executing (CREATE or UPDATE) but this can be modified or even omitted. Thus, LOCKSMART can operate using only program names (and without process names) but this typically degrades usefulness of resultant metrics since a single program might need to lock the same data set repeatedly for different operations, each of which might on average might take varying amounts of time to complete. Thus, use of &LOCKTRACKPROCESS is advised to better organize metrics that are demonstrated in the next section. The previous code generates the LOCKNTRK.Control control table (if it does not already exist) and creates ten entries for both the CREATE and UPDATE processes. Up until this point, the LOCKSMART macro has only demonstrated

equivalent functionality to LOCKANDTRACK but, in the next section, the added benefits and functionality of LOCKSMART are demonstrated.

PURGING AND METRICS GENERATION

On a regular basis, the LOCKNTRK.Control control table will need to be both purged and analyzed. Purging removes older entries to ensure that the table does not become unwieldy (and to ensure that the entries remain recent and relevant), while analysis provides metrics that LOCKSMART utilizes to predict file access wait times. Both purging and analysis are highly customizable, but for a proof of concept, the following code purges all performance data older than seven days and creates a threshold value equal to the mean number of seconds that a specific process (within a specific program) locked a specific data set. In other words, if the PERM.Critical data set is locked on average for 61 seconds by the CREATE macro in the Locktest.sas program, the following PURGE macro will calculate that mean and set it as the threshold to be used in subsequent calls to LOCKSMART when the identical program, process, and data set are involved:

```
* THIS PROGRAM SHOULD BE SAVED AS A NAMED FILE SUCH AS C:\PERM\PURGE.SAS;
libname perm 'c:\perm';
libname lockntrk 'c:\perm\lockntrk';
%include 'c:\perm\lockandtrack.sas';

* sample macro that purges old performnace data and creates metrics;
%macro purge(days= /* days to retain historical performance data */,
  sample= /* minimum sample size for which to use performance data */);
%lockitdown(lockfile=lockntrk.control, sec=1, max=600, type=W,
  canbemissing=NO);
data lockntrk.control (drop=days);
  set lockntrk.control;
  days=round((datetime()-dtg_request)/86400);
  if not missing(days) and days<=&days;
run;
proc sort data=lockntrk.control;
  by lib fname program process type dtg_request;
run;
%lockitdown(lockfile=lockntrk.metrics, sec=1, max=600, type=W,
  canbemissing=YES);
data temp (drop=dtg_lockstart dtg_lockstop);
  set lockntrk.control (keep=lib fname program process type dtg_lockstart
    dtg_lockstop);
  if not missing(dtg_lockstart) and not missing(dtg_lockstop) then
    locksecs=dtg_lockstop-dtg_lockstart;
  if not missing(locksecs);
run;
proc means data=temp noprint;
  output out=lockntrk.metrics n=n mean=thresh;
  by lib fname program process type;
  var locksecs;
run;
data lockntrk.metrics;
  set lockntrk.metrics (drop=_type_ _freq_);
  thresh=round(thresh);
  if n>=&sample;
run;
&lockclr; * releases the METRICS lock if it was required;
lock lockntrk.control clear;
%mend;
```

```
%purge(days=7, sample=5);
```

Thus, in this example, because two processes exist—CREATE and UPDATE—within the Locktest.sas program, only two entries were created in the LOCKNTRK.Metrics data set, as demonstrated in Table 1. However, as the number of programs and subordinate processes increases and as these interact with an increasing number of data sets, the number of entries will grow exponentially. For example, if Locktest.sas were subsequently run on a separate data set (i.e., not PERM.Critical), the number of observations in the LOCKNTRK.Metrics control table would double because new entries would be created for the new data set for both the CREATE and UPDATE processes. The N column represents the number of lock attempts that were recorded within the specified timeframe (7 days) dictated by the purge.

For example, the first line of Table 1 can be read as “In the last seven days, the CREATE process within the Locktest.sas program required an exclusive lock on the PERM.Critical data set ten times during which the data set was locked an average of 61 seconds.”

lib	fname	program	process	type	n	thresh
PERM	CRITICAL	LOCKTEST.SAS	CREATE	W	10	61
PERM	CRITICAL	LOCKTEST.SAS	UPDATE	W	10	106

Table 1. Sample LOCKNTRK.Metrics Performance Metrics

Note that because the possibility exists that either the LOCKNTRK.Control or LOCKNTRK.Metrics data set will be in use, and because the PURGE macro requires exclusive locks on both of these control tables, the LOCKITDOWN macro is invoked twice within PURGE to ensure that file access collisions do not occur. The PURGE macro demonstrated is a proof of concept, intended to be modified for the specific needs of a user of team. For example, in addition to modifying the parameter values for DAYS and SAMPLE, the algorithm itself could be modified so that the THRESH value created is not the mean but instead the median, the mean plus one standard deviation, or some other metric. If the LOCKSMART macro (or LOCKANDTRACK) is being used, the PURGE macro should ideally be scheduled so it can run automatically on a recurring basis, for example nightly, weekly, or monthly.

If PURGE is not run on a regular basis, the data in LOCKNTRK.Metrics can become stagnant or inaccurate. For example, if the PERM.Critical data set is consistently increasing in size over time, the amount of time required to run CREATE or UPDATE is also likely to increase commensurately over time. If this occurs, more recent metrics for file access times will be more accurate than older metrics because newer metrics will depict the current file size and associated file lock times, whereas older metrics might inaccurately portray faster execution times that occurred on smaller (now outdated) data sets. Even when data set file size is relatively stable over time, running PURGE will remove older observations in LOCKNTRK.Metrics that can cause the control table to become too large and thus slow performance.

FINALLY RUNNING LOCKSMART

The LOCKSMART macro has already been used to build a control table of historical metrics on the CREATE and UPDATE processes, but its further use can now be demonstrated. Continuing this example, an additional FREQ process might generate frequencies on the PERM.Critical data set, thus requiring a shared lock on the data set. The following code can be saved as the SAS program C:\perm\freq.sas and executed. Again, because LOCKSMART must be run from a named SAS program, the following code must be saved before it is executed:

```
* THIS PROGRAM SHOULD BE SAVED AS A NAMED FILE SUCH AS C:\PERM\FREQ.SAS;
libname perm 'c:\perm';
%include 'c:\perm\locksmart.sas';

* simulates an analytic module;
%macro freq();
%let locktrackprocess=&sysmacroname;
```

```

%locksmart(lockfile=perm.critical, sec=5, max=20, type=R, canbemissing=NO);
%if %length(&locktrackerr)=0 %then %do;
    proc freq data=perm.critical;
        tables char1;

    run;
    %end;
%locksmart(lockfile=perm.critical, remove=Y);
%mend;

%freq;

```

Note that because this code is viewed as less critical than the CREATE or UPDATE modules, the SEC parameter is increased to five seconds, thus giving precedence to other processes that might be more important (as indicated by a SEC of less than five seconds). In a production environment, the SEC value might vary from one to 60 (or even higher) to ensure the importance (and prioritization) of critical processes is incorporated.

When the code is executed, several outcomes could occur, some of which include:

- LOCKSMART determines that the data set PERM.Critical is not locked (or another process only has a shared lock), thus enabling FREQ to complete immediately. The LOCKNTRK.Control control table is updated, the FREQ module executes, and the control table is updated again following FREQ to indicate how long FREQ required access to the data set.
- LOCKSMART determines that the data set PERM.Critical is locked exclusively but no relevant entry exists in the control table (indicating that a person or program has locked the data set but not utilized LOCKSMART or LOCKANDTRACK to do so). LOCKSMART enters a busy-waiting cycle for 20 seconds, attempting to access the data set every five seconds, and will either gain access (and complete the FREQ module) or fail to gain access (and skip the FREQ module). The control table is updated with the success or failure and records that FREQ was delayed (or blocked entirely) by an unknown process, thus demonstrating to a SAS administrator the need to more thoroughly implement LOCKSMART for all accesses to the data set PERM.Critical.
- LOCKSMART determines that the data set PERM.Critical is locked and, because the data set was locked with either LOCKSMART or LOCKANDTRACK, finds the corresponding lock entry in the control table. LOCKSMART then attempts to locate a matching lock entry in the LOCKNTRK.Metrics metrics table but, if it cannot, LOCKSMART enters the busy-waiting cycle for 20 seconds. The control table is updated with the success or failure. This occurs when a separate process did utilize LOCKSMART to lock the data set PERM.Critical but when insufficient historical data exist (established by the SAMPLE parameter in the PURGE macro) to generate metrics.
- LOCKSMART determines that the data set PERM.Critical is locked and, because the data set was locked with either LOCKSMART or LOCKANDTRACK, finds the corresponding lock entry in the control table. LOCKSMART then attempts to locate a matching lock entry in the LOCKNTRK.Metrics metrics table and, if successful, knows not only the program and process that has locked the data set, but also when it was locked, and for how long on average the lock will persist. With this knowledge, LOCKSMART is able to decide autonomously (based on the MAX parameter threshold) whether to stick around and wait for the process to complete (and unlock the data set) or whether to abort and do some other task.

GET_PROGRAMNAME MACRO

The GET_PROGRAMNAME macro retrieves the file name of the saved file that the SAS application is currently executing. The macro is the reason that LOCKSMART and LOCKANDTRACK must be run on named rather than untitled SAS programs. As written, the macro is portable to the SAS Display Manager (for Windows), SAS Enterprise Guide (for Windows), and the SAS University Edition. Because LOCKANDTRACK was not portable to Enterprise Guide, this improvement extends the functionality of LOCKSMART to SAS Enterprise Guide. However, because the macro has only been tested in these environments, modifications might be necessary to port to other environments.

```

%macro get_programname;
%global sys;
%let sys=;
%global locktrackprogram;
%let locktrackprogram=;
%if %symexist(_clientprojectname) %then %do;
    %let sys=WIN;
    %let locktrackprogram=%sysfunc(dequote(&_sasprogramfile));
%end;
%else %if &SYSSCP=WIN %then %do;
    %let sys=WIN;
    %let locktrackprogram=%sysget(SAS_EXECFILENAME);
%end;
%else %if &_CLIENTAPP=SAS Studio %then %do;
    %let sys=UNIX;
    %let locktrackprogram=&_SASPROGRAMFILE;
%end;
%else %do;
    %put Environment Not Defined!;
    abort abend;
%end;
%mend;

```

The GET_PROGRAMNAME macro is utilized by LOCKSMART and is included in Appendix A.

CONCLUSION

The LOCKSMART macro includes all functionality of LOCKANDTRACK as well as LOCKITDOWN but builds upon these previous versions by also enabling autonomous decision making. Through automated real-time generation, collection, and analysis of performance metrics, LOCKSMART can flexibly respond to shifting dynamics of the production environment, enabling program flow to identify bottleneck processes and route around them. Rather than painfully and inefficiently waiting for busy-waiting spinlock cycles to complete—or, worse yet, time out—LOCKSMART can avoid this congestion and quickly get processes back on the data superhighway.

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).
- ⁱⁱ Hughes, Troy Martin. 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*. 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

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX A. LOCKSMART MACRO

```

* The GET_PROGRAMNAME and LOCKSMART macros can be saved into;
* a single SAS program (C:\perm\locksmart.sas) to mirror this text;
* but can be moved to any location with modification of the;
* LIBNAME and INCLUDE statements, as indicated below;

%macro get_programname;
%global sys;
%let sys=;
%global locktrackprogram;
%let locktrackprogram=;
%if %symexist(_clientprojectname) %then %do;
    %let sys=WIN;
    %let locktrackprogram=%sysfunc(dequote(&_sasprogramfile));
%end;
%else %if &SYSSCP=WIN %then %do;
    %let sys=WIN;
    %let locktrackprogram=%sysget(SAS_EXECFILENAME);
%end;
%else %if &_CLIENTAPP=SAS Studio %then %do;
    %let sys=UNIX;
    %let locktrackprogram=&_SASPROGRAMFILE;
%end;
%else %do;
    %put Environment Not Defined!;
    abort abend;
%end;
%mend;

%macro locksmart(lockfile= /* data set in LIBRARY.DSN or DSN format */,
    sec=5 /* interval in seconds after which data set is re-tested */,
    max=300 /* maximum seconds waited until timeout */,
    type=W /* either (R) or (W) for READ-only or read-WRITE lock */,
    canbemissing=N /* either (Y) or (N), 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 'c:\perm\lockntrk'; * MUST BE MODIFIED;
%end;
%include 'c:\perm\lockitdown.sas'; * MUST BE MODIFIED;
%local i;
%local lib;
%local tab;
%local starttime;
%local max;
%local loc;
%local lockprogram;
%local lockprocess;
%local secondslock;
%local thresh;
%local quitlock;
%let quitlock=;

```

```

%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 datetimel7.;
  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=;
%get_programname;
%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 - 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 - no library %upcase(&lib);
  %goto err;
%end;

%if %sysfunc(exist(&lib..&tab)) ^ 1 %then %do;
  %if &canbemissing=N %then %do;
    %let locktrackerr=LOCKITDOWN failed - %upcase(&lib..&tab)
          does not exist;
  %end;
%end;

```

```

        %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 - TYPE must be either R or W;
    %goto err;
%end;
%if &canbemissing^=Y and &canbemissing^=N %then %do;
    %let locktrackerr=LOCKITDOWN failed - 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) or
&quitlock=YES);
    %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;
    %if %length(&quitlock)=0 %then %do; * evaluate whether to stay or
        abort only the first cycle;
        * determine what program/process has the data set locked;
        %lockitdown(lockfile=lockntrk.control, sec=1, max=60,
            type=W, canbemissing=NO);
        %let lockprogram=;
        %let lockprocess=;
        %let secondslock=;
        data _null_;
            set lockntrk.control;
            if lib="%upcase(&lib)" and fname="%upcase(&tab)" and
                not missing(dtg_lockstart) and
                missing(dtg_lockstop) then do;
                call symput('lockprogram',strip(program));
                call symput('lockprocess',strip(process));
                call symput('secondslock',
                    strip(put(round(datetime()-
                        dtg_lockstart),8.)));
                end;

            run;
            &lockclr;
            * determine how long the data set will be locked;
            %lockitdown(lockfile=lockntrk.metrics, sec=1, max=60,
                type=W, canbemissing=NO);
            %let thresh=;
            data _null_;
                set lockntrk.metrics;
                if lib="%upcase(&lib)" and fname="%upcase(&tab)" and
                    program="&lockprogram" and
                    process="&lockprocess" then do;
                    call symput('thresh',strip(put(thresh,8.)));
                end;

            run;
            &lockclr;
            * do not abort if historical data not in metrics table;
            %if %length(&lockprogram)=0 or %length(&thresh)=0
                %then %let quitlock=NO;
            %else %if %sysevalf(&thresh-&secondslock>&max)
                %then %let quitlock=YES;
            %end;

```

```

        %else %do;
            %let dsid=0;
            %put SLEEPING &sys;
            %let sleeping=%sysfunc(sleep(&sec));
            %end;
        %end;
    %end;
%if &dsid=0 %then %do; *timeout or abort before lock achieved;
%if quitlock=YES %then %let locktrackerr=LOCKSMART aborted attempting
    to gain %sysfunc(ifc(&type=W,an exclusive,a shared)) lock on
    &lockfile because &lockprogram &lockprocess was anticipated to
    have the data set locked an additional
    %sysevalf(&thresh-&secondslock) seconds;
%else %let locktrackerr=LOCKSMART 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;
```