Enhancing SAS/SHARE® Software Capability: Data Set Control, Locking, and Enqueueing
Brian Perkinson, SAS Institute Inc., Cary, NC

ABSTRACT

SAS/SHARE® software allows two or more users to update a SAS® data library simultaneously. The key to SAS/SHARE software is a separate SAS execution called the server, which performs read and write operations on behalf of the users who are sharing a library. When a read or write request from a user conflicts with ongoing activities of other users, the server rejects the request and the user’s PROC or DATA step does not execute. SAS/SHARE software itself offers no way to avoid these conflicts. However, sites licensing SAS/SHARE software can use exits from the SAS System to implement SAS source statements that enable a SAS program to check for conflicts before they occur or to wait until the required data set is available for processing. These statements enable a SAS programmer to preserve the natural design of an application and take full advantage of SAS/SHARE software without risking the integrity of the data involved.

INTRODUCTION

The most commonly asked how-to question regarding SAS/SHARE software has to do with the CONFLICTING LOCK messages that are issued when, for example, you attempt to replace a data set in a DATA step while someone else is using the F$EDIT procedure to edit it. Users want to know if they can avoid these messages or at least detect the error and recover from it in some way that is transparent, or at least friendly, to the user. The answer is a qualified no or a qualified yes, depending on how you look at it.

Two Solutions to the Conflict Problem

The approaches to the conflict problem fall into two basic categories: detecting that the conflict has occurred and handling it in some way (such as retrying the failing program step, issuing a message, or giving an interactive user the option of trying again immediately or starting over later), and avoiding the conflict altogether. In Release 5.16 of the SAS System, there was no straightforward way for a program to determine that conflict had occurred.

However, for applications in which the SAS System was running interactively under the SAS Display Manager System, there was a fairly ingenious solution. The typical use of this solution was on SAS/SHARE® program screens. The program screen would contain the potentially failing PROC or DATA step. After this step, the %DMSCMO macro was used to execute a FIND command for the CONFLICTING LOCK message on the log screen. If this text was found, a message to that effect would be placed in the automatic DMS macro variable &DMSLMMSG to be displayed under the log screen command line. The program would then check the value of &DMSLMMSG and proceed accordingly. This was a clever way of determining when conflict had occurred and it seems to have worked reasonably well. However, batch and other noninteractive programs, as well as applications in which the SAS System runs interactively in line mode, were still subject to undetectable conflicting lock errors that made application performance undependable.

With Release 5.18 of the SAS System, there is a new automatic macro variable called &SYSERR. This macro variable contains the condition code from the most recently executed PROC or DATA step. This makes detecting the conflicting lock error much easier. If a PROC or DATA step fails to execute because of a conflicting lock, &SYSERR is set to a nonzero value. So you simply test &SYSERR after the program step of interest and, if it is nonzero, handle the error in some appropriate manner. However, this method still only works for interactive applications. When it is running noninteractively, the SAS System goes into syntax-check mode when the error is encountered and subsequent steps are not executed.

AVOIDING THE CONFLICT

Since the idea of detecting the conflict after it has occurred and somehow handling it only serves well for interactive applications, avoiding the conflict altogether now seems a more attractive alternative. More accurately, we want to prevent conflicts from reaching the server, which necessarily has a rather single-minded approach to handling conflicts (that is, the server always rejects new requests that conflict with ongoing activities). Instead, we would like to detect the conflict ourselves and handle it in some more flexible manner. Put simply, we want to do our own file locking.

Although this may sound difficult or complicated, it doesn’t have to be. In fact, there is a facility within MVS that does just what we want. We want to be able to request a certain type of control, or lock, on a data set. Under certain circumstances, we want to be able to wait if the data set is unavailable. Other times we want to be able to detect that the data set is not available and continue processing. Handily, this is just what the global resource serialization (GRS) component of the operating system does. GRS processes requests for control of any resource used by more than one process. So, all we have to do is call GRS and pass the appropriate information. The advantage of this is that the locking is fairly simple to understand (there are only two kinds, as we’ll see) and to accomplish. The disadvantage is that calling GRS requires some programming in another language besides the SAS programming language (this paper uses assembly language). Therefore, this is not the solution for every site. But if you have someone to do the programming, this solution can be implemented fairly easily and it does work pretty well.

GRS

The key to understanding the method of locking described in this paper is understanding GRS. Calls to GRS from an assembler program are made using two IBM®-supplied assembler macros, ENQ and DEQ. ENQ is used to request control of a resource (enqueue on it). DEQ is used to relinquish control of a resource (dequeue from it). When you use the ENQ macro, you must specify the resource you want control of, the type of control you want, and what you want to do in the event that you cannot have control immediately. For our purposes, the resources we will request control of will be SAS data sets.

Requesting Control of a Resource: ENQ

Resources are represented to GRS by two names, the name and the name. The name is a string 1 to 8 characters in length. The name is a string 1 to 255 characters long. These names are sometimes referred to as the major name and minor name,
respectively. When you use the ENQ (or DEQ) macro, what you supply (or these names in mostly up to you, there are certain
quantities that you cannot use). GRS isn't interested in what the
resource is; it's only interested in the resource's name. The only
ting to remember is to be consistent. Processes competing for
the same resource (interactive applications, batch programs, and
so on) must always use the same name and name to represent a
given resource.

There are two kinds of control that you can request for a given
resource: shared and exclusive. And there are two kinds of
requests that we will consider those that indicate that you want
to wait for an unavailable resource and those that indicate that
you want to continue executing immediately in the event that you
cannot get control.

Let's consider an example. Suppose we want exclusive control
of a sequential data set, and we want to wait until such control
is available. Now, we can think of a file being available and that you
specify a name of 'INVOICE' and an arbitrary chosen string. So our ENQ specification in pseudocode looks like this:

ENQ QNAME='INVOICE'
  NAME='INVOICE INFO'
  CNTL=EXCLUSIVE
  RET=NONE

When we make this call to GRS, it searches its lists of existing
enqueues for those with a matching name and name. Because we
have requested exclusive control, if there are any existing
enqueues at all, either shared or exclusive, we will not be able
to get control of the desired data set immediately. And if that is
the case, because we specified RET=HAVE, we will wait until we
do get control of the data set before we continue processing
(GRS puts us in a "wait state").

Rather than using an arbitrarily chosen string for the qname, we
can choose something more meaningful, something that
describes the kind of processing, the reason we want control, the
application that is doing the enqueue, or just the file itself. For
now, let's change our qname to 'INVOICE' and let's say we don't
want to wait if the file is not immediately available. Now our ENQ
specifications looks like this:

ENQ QNAME='INVOICE'
  NAME='INVOICE INFO'
  CNTL=EXCLUSIVE
  RET=NONE

Because we've specified RET=USE, if the resource represented
to GRS by the names 'INVOICE' and 'INVOICE INFO' (and remember, GRS doesn't know or care that these names represent
a file) isn't available, the ENQ macro returns a code of 4.

When we issue one of the ENQs shown earlier, GRS first looks
for existing enqueues with a matching name. If any are found,
GRS looks among them for any with a name of 'INVOICE INFO'.
If any enqueues with name and name matching ours are found,
we cannot process the file immediately because we require
exclusive control of it. If we have issued the first ENQ shown ear-
erlier, our enqueue is added to the list, and when all preceding
enqueues have been released (via DEQ), we will resume execu-
tion and process the invoice information. If we have issued the
second ENQ shown earlier, GRS returns to us immediately with
a code of 4, meaning we could not obtain an exclusive lock on
the file. We would presumably check this return code and handle
the situation in some appropriate manner.

If we issue our ENQ with CNTL=SHARED instead, then the file
would only be immediately unavailable if some other process had
an exclusive lock on it. As before, if we have specified
RET=HAVE, GRS waits until the exclusive lock has been
released before returning to us to indicate that we have the con-

When we issue our ENQ other processes already have a
shared lock on the file, then our ENQ is added to that list, ahead
of any outstanding exclusive ENQs, and we would continue with
whatever processing required shared control of the file.

Representing the Resource

Now consider our choice for name. This is one of the names
used to represent to GRS the file we are interested in. The file
contains invoice information, so 'INVOICE INFO' is a natural
choice, and fairly descriptive. However, suppose some other pro-
cess or application needs control of another file that also contains
invoice information. In that process issues its ENQ with name of
'INVOICE INFO' (and a name of 'INVOICE') to obtain exclu-
sive control of its file and we subsequently issue our ENQ as
shown above, our file will appear unavailable when it is not. In
other words, our processing might wait unnecessarily on another
unrelated process. Or, if we have indicated that we do not want
to wait, our request for control of our file is rejected immediately
when it is actually available.

Also consider the possibility that some other process, which may
or may not be part of our application, may need to access the
file we intend to operate on. If that process is interested not in
the invoice information as a whole but just in the customer names
in the file, it may seem natural for that process to use an name of
'CUSTOMER NAMES' for its ENQ. However, this name does
not match ours, although it is intended to represent the same
resource. In this case, we may process the invoice file under the
incorrect understanding that we are the only ones accessing it.
In other words, we are prevented from processing this file in
some manner that requires exclusive control of it when others
are already accessing it. More importantly, others are not pre-
vented from processing the file while we are already doing so,
which can be a real problem if their processes change the file.

Thus we need to choose for the name something descriptive
and something that is not only grounded in the identity of the resource
but is also unique to the resource we mean to refer to (in our
equivalents so far, a sequential file containing invoice information).
A logical choice that fits these criteria is the physical file name
or some string containing the physical file name.

Releasing the Resource: DEQ

Once you've done with the resource you've obtained control
of via ENQ, you must release it so that it will be available to other
processes. The DEQ macro is used to relinquish control of a
resource requested via ENQ. When you use the DEQ macro, you
must specify the resource you want to release in the same way
that you specified the resource on the ENQ macro. It's not neces-

Locking SAS Data Sets via GRS

Now that we've seen how GRS works, how do we apply it to lock-
ing SAS data sets? The first task is to decide how to represent
the SAS data set to GRS. For name, we want to use not only the
SAS data set name but the physical name of the MVS data
set that is the SAS data library as well. If we use just the SAS
data set name, we could have problems similar to the ones described earlier if we have data sets with the same name in different libraries. If we use just the DSNAME of the library, we could have problems if the library contains more than one data set. Of course, the syntax here is entirely up to us; for the purposes of this paper, we will use this familiar format, resembling a partitioned data set and member name specification, to refer to the SAS data set INVOICE in the data library whose DSNAME is 'SHIP,ACCT,HIST'.

*SET, ACC, Hist(INVOICE)*

For qname, we should choose something indicative of the resources we are looking for and the kind of processing we are doing. In our examples, we use a qname of 'SAS' because the resources we are processing are SAS data sets and we use SAS software to process them. Remember that both qname and name can be anything you choose as long as they are sufficient to represent the particular resource you intend to process and you use them consistently.

Now that we've established the names we will use to represent the data sets we will process, we need to decide what kind of lock to request for the different ways of processing SAS data sets. Rather than be concerned with whether we will update a SAS data set or just read it (which is the most common criterion for determining what kind of lock is needed for any resource), our rule of thumb will be as follows:

*Any activities that the server allows to occur simultaneously will require shared control. Any activities that conflict with other activities will require exclusive control.*

This means that for a DATA step to read a data set using a SET statement with CNTLLEV=BLK specified, we need shared control. Likewise, for an input data set to PROC COPY, for the BASE = data set on a PROC APPEND statement, and for PROC FSEdit, we also need shared control because none of these activities conflicts with each other. For a data set named on a SET statement without the CNTLLEV specification or on a DATA statement, we need exclusive control. This is really just common sense; our objective is to supplement the server's locking with our own so as to prevent conflicts from reaching the server. Therefore, our only interest is to catch any processing that will conflict with ongoing activities so that it is not unilaterally rejected by the server.

One benefit of using our own locks to control access to the data and prevent conflict is that we can not only obtain a lock (or fail to) prior to the PROC or DATA step that will actually use the data set, but we can also hold the lock across steps. This ensures that for multiprocess processing, if any of the steps executes, all will execute. We will not be vulnerable to having part of a process run successfully and part rejected for conflicting locks.

Calling GRS from a SAS Program

So now, except for the particulars of specifying the ENQ and DEQ macros from an assembler program, we are ready to make our calls to GRS to request and relinquish control of the SAS data sets we will process. But how do we make these calls from a SAS program? One obvious way is to execute these assembler programs from within the SAS System with the CALL command or as TSO commands themselves. However, the only way to execute TSO commands from a SAS execution is with the TSO statement (or the %TSO statement, TSO( ) DATA step function, and TSO Display Manager command). And the TSO statement is ignored when the SAS System is running in batch, meaning our enqueues wouldn't be executed from batch SAS programs. Even if this wasn't a problem, enqueues are handled by GRS on a task basis, and when a task with outstanding enqueues terminates, those enqueues are released. And the TSO statement (and the %TSO statement and the TSO( ) DATA step function) executes TSO commands by attaching them as separate tasks. This means our enqueuing module would be attached as a task separate from the SAS task, issue its enqueue, and then terminate, causing the just-issued enqueue to terminate as well. Therefore, we must look for alternatives that don't have these problems.

Another method of executing user-written programs (written in languages such as assembler) is the PROC statement. But whether we write our programs as SAS procedures, following the guidelines in the SAS programmer's guide, or execute them as external programs using the external form of the PROC statement, parameter passing and parsing may be somewhat more difficult or involved than we'd like.

We could also write these programs as DATA step functions or CALL routines, but then calling them would require the execution of a DATA step, which definitely involves overhead we would rather avoid.

The remaining commonly used method of executing user-written assembler programs is the SAS System user exits. An exit is a predefined point in the execution of a program or system at which a routine can gain control for processing not directly related to the business at hand. For example, there is an exit from the SAS System called the step accounting exit. This exit allows a site-supplied routine to execute for the purpose of performing SAS program step accounting using information supplied to the exit routine such as the procedure name and the amount of CPU time used by the procedure. Fortunately for us, there is an exit that is perfectly suited to our needs. This exit, the unrecognized statement exit, allows site-supplied logic to get control whenever the SAS System encounters a statement that it does not recognize. This exit is the perfect means of creating your own statement for use in SAS programs. Thus, we can use ENQUEUE and DEQUEUE statements that cause an exit routine of our creation to get control and invoke our enqueuing and dequeuing programs. This method works equally well for all applications, whether batch or interactive. And we can use standard Institute-supplied macros for parsing these statements so that the programmer can pass the appropriate parameters to our enqueue routine to indicate what kind of ENQ should be issued.

SAS System User Exits

When one of the programs that make up the SAS System wants to take an exit, it loads the number of the exit in general purpose register 0 (expressed as "RD") and branches to the entry point in the module NASUSER. The main NASUSER logic saves the caller's registers and then uses the value in RD as the offset into a branch table to branch to the location in NASUSER that represents the user exit. An exit is turned on by changing the return source and then reassembling and relinking the module. The exit routine itself either follows the NOP "in-line" or is contained in a site-supplied module with a pre-established name.

Implementing the ENQUEUE and DEQUEUE Statements

The first step in our exit logic is to determine if one of the statements we are implementing has been encountered. When the unrecognized statement exit is taken, the address of the SAS System's "word queue" is loaded into R7. The first word in the queue is in the 200-bytes field labeled W1. If the first word of the statement is ENQUEUE or DEQUEUE (and we could just have used LOCK and UNLOCK or any such appropriate pair),
then we want to call the appropriate routine. If not, then we simply return control to the SAS System, which will issue a standard syntax error message.

We can call our routines (in modules with names ENQUEUE and DEQUEUE) by using the Institute-supplied macro SASLINK. This macro takes the module to be called as its parameter and generates a SASLOAD of the module, a branch and link to it, and a SASDLT of the module. SASLOAD and SASDLT are Institute-supplied macros that take a module name as their parameter. SASLOAD loads the module and SASDLT deletes the module. However, if we SASLINK the appropriate module every time one of our statements is used, we will be loading and deleting these modules repeatedly. Instead, we can load them at SAS System initialization, save the addresses at which they are loaded, and simply branch to them when we need them. Then we can delete them at SAS System termination, improving the efficiency of our exit processing. We can use the SAS System initialization and termination user exits to load and delete our modules.

The ENQUEUE Statement Routine

In our ENQUEUE statement routine, we must parse the ENQUEUE statement and, if there are no severe errors, issue the appropriate ENQ. Before we can know how to parse it, we must decide on the syntax of the statement. We must have the qname and name; we will use positional parameters for them, with qname specified first. We also may have the CNTL and RET—parameters, and keyword parameters seem a natural choice for these parameters, and keyword parameters seem a natural choice for these parameters, and keyword parameters seem a natural choice for these parameters. However, notice that what a parameter following the keyword parameter can be is not immediately available might look like this:

```
ENQUEUE SAS 'SHIP.ACCT.HIST(INVOICE)' CNTL=SHARED RET=HAVE;
```

Since the qname and name are positional, we can check for them first. We use the SAS System's WORD routine (called via the Institute-supplied WORD macro) to advance the word queue. Now the qname, the second word in the statement, is the first word in the queue. Since there is no real validation to be done for the qname value, we can store this parameter in the appropriate place. Or, if we want to be more sophisticated, we could ensure that it is not longer than 8 characters and that it is not one of the keywords CNTL- and RET-. The very least we should do is ensure that there actually is a parameter following the ENQUEUE keyword, and not just a semicolon, since the qname is required.

Our processing for the name parameter follows the same pattern. Although the ENQ macro name parameter can be up to 256 characters in length, the SAS System word queue allows only 200 characters per word, so that will be the maximum for the name parameter in our ENQUEUE statement. The value in the example above is specified in quotes because the WORD routine recognizes periods and parentheses as "word" separators, or delimiters, but treats an entire quoted string as a single word (which is what we need). For the CNTL- and RET—parameters, we must check for the proper keywords, the equals sign (=), and an acceptable value. We use these values to determine what kind of ENQ to issue.

The DEQUEUE Statement Routine

The processing for the DEQUEUE statement routine is similar to that for the ENQUEUE statement routine since the two parameters (qname and name) are the same.

Passing the Return Code

The only piece missing from the puzzle now is how we can pass the return code from the ENQ macro back to our SAS application. If we were executing our enqueuing routine with the SASI macro, we could have the routine set its return code in R15 before returning to the SAS System, and the SAS macro facility would set this return code in the macro variable &SYSRC automatically. However, because we are using the unrecognized statement exit, we need some other method. It would be nice if we could set the return code in &SYSRC or some other agreed upon macro variable, but there is no documented method of doing so. However, we are in luck. The automatic macro variable &SYESERR, new with Release 5.18 of the SAS System, is a little different from other macro variables. This variable is actually a 4-byte character value (for example, X'FOFRPS4' for the string "WON"), to indicate an error). Then our SAS application can simply test the value of &SYESERR just as it would after a PROC or DATA step.

Examples Using the ENQUEUE and DEQUEUE statements

Let's look at some typical kinds of SAS processing using our new statements. In the example below, we want the interactive user to edit the data set INVOICE via PROC FSEDIT. Since multiple users can edit the data set simultaneously via the server, we indicate CNTL—SHARED. We also specify RET—HAVE so that if some other process has exclusive control of the data set, execution will be suspended until INVOICE is available. As a result, we should never receive a nonzero return code from the ENQUEUE statement, so there is no need to check &SYSRC.

```
LIBNAME SHIPRE 'SHIP.ACCT.HIST(SERVER-SASH);
ENQUEUE SAS 'SHIP.ACCT.HIST(INVOICE)' CNTL=SHARED RET=HAVE;
PROC FSEDIT DATA=SHIPRE.INVOICE user=SHIPRE.INVOICE; SUB;
ENQUEUE SAS 'SHIP.ACCT.HIST(INVOICE)';
```

Notice that the DSNAME of the SAS data library appears not only on the LIBNAME statement but also on each ENQUEUE and DEQUEUE statement that references a data set in that library. Hard-coding the library name as shown above is probably not good programming practice for large applications (that's why DDNAMES were invented to begin with). So for such applications, a better method is to set a macro variable to the library DSNAME and refer to that macro variable on the LIBNAME statement and the ENQUEUE and DEQUEUE statements, as shown below:

```
LIBNAME SHIPRE 'SHIP.ACCT.HIST';
LIBNAME INVOICE "SHIIPBIB\INVOICE\SHIPRE.INVOICE" SHIPRE-SASH;
ENQUEUE SAS 'SHIPRE(INVOICE)' CNTL=SHARED RET=HAVE;
PROC FSEDIT DATA=SHIPRE.INVOICE user=SHIPRE.INVOICE; SUB;
ENQUEUE SAS 'SHIPRE(INVOICE)';
```

Finally, notice that the EXIT routine cannot call the ENQUEUE and DEQUEUE statements to edit the same data set, but we want to allow the user to continue with other processing if he or she cannot do so immediately. So we specify RET—USE and test the value of &SYESERR afterward. Notice that the DEQUEUE, as well as the PROC FSEDIT step, is conditional on the success of the ENQUEUE.
The example below uses a DATA step to replace the data set INVOICE. Because this requires exclusive control of the DATA step and because users are likely to be editing this data set for long periods of time, the data set may be unavailable for a longer time than an interactive user is willing to wait. Rather than tie up the user's terminal waiting for this DATA step to run, we specify RET=USE so that we can issue an explanatory message and allow the user to continue with other work.

```
ENQUEUE SAS "INVOICE1.INVOICE" CNTL=EXCLUSIVE RET=USE;
S1P (ASTSERR=0004) S1RE
   ENQ;
   DATA SHIPMST.INVOICE;
   SET SHIPMST.INVOICE;
   * More DATA step statements ;
   .
   RUN;
ENQUEUE SAS "INVOICE1.INVOICE";
S1RE;
S1LE / (ASTSERR=0004, i.e., ENQUEUE failed) */
S1P;
S1LE CANNOT UPDATE INVOICE DATA SET NOW, TRY LATER;
S1RE;
```

On the other hand, if we were going to run this DATA step in a batch job, we may prefer to have the job wait for the data set rather than forcing the user to resubmit the job more than once. To do this, we would specify RET=HAVE and we would not need to test ASYSERR afterward.

In this final example, the user will edit the data set and we will then post-process and replace it in a DATA step. Although the PROC FSEDIT step requires only shared control of the data set, the DATA step requires exclusive control. Since we don't want to do the PROC FSEDIT step if we cannot do the DATA step and since we are unwilling to force the interactive user to wait until the DATA step has completed, we specify CNTL=EXCLUSIVE and RET=USE prior to the PROC FSEDIT step. Then if we are able to get the lock we need, we hold it across the step boundary until after the DATA step has completed.

```
ENQUEUE SAS "INVOICE1.INVOICE" CNTL=EXCLUSIVE RET=USE;
S1P (ASTSERR=0004) S1RE
   ENQ;
   PROC FSEDIT DATA=SHIPMST.INVOICE SCAPE=SHIPMST.INVOICE;
   RUN;
   DATA SHIPMST.INVOICE;
   SET SHIPMST.INVOICE;
   * More DATA step statements ;
   .
   RUN;
ENQUEUE SAS "INVOICE1.INVOICE";
S1RE;
S1LE / (ASTSERR=0004, i.e., ENQUEUE failed) */
S1P;
S1LE CANNOT UPDATE INVOICE DATA SET NOW, TRY LATER;
S1RE;
```

A WORD OF CAUTION

Although implementing your own data set locking does require some nontrivial planning and programming, it can be made to work very well. However, as we saw with our first example, every application that processes a data set must first use ENQ to request control of it and must use the same combination of name and name. Otherwise, applications that depend on this supplemental locking to ensure their successful processing run the risk of failing because a conflict isn't detected before it reaches the server. If you can ensure that whoever writes programs to access the SAS data sets you are locking uses the ENQUEUE statement first, then this is a workable and reliable solution.

SAS, SAS/AF, and SAS/SHARE are registered trademarks of SAS Institute Inc., Cary, NC, USA.

IBM is a registered trademark of International Business Machines Corporation.