The Fox Hunt

Tracking User Logins

By Whil Hentzen

The email has been rolling in as a result of the last few months columns on handling user issues, so this month, we're going to do an encore performance. A number of people have asked seemingly unrelated questions that all have the same answer. For example, one question is "I've got a user who swears they're not rebooting the system, but every time I look in their directory, I find more "*.TMP" files left over from FoxPro not shutting down properly. What do I do?" Another question is "I had a user swear that they were testing the system, but two weeks later, a screen crashed and they claimed that something had been changed. I'm sure that they had never looked at that screen. How do I prove it?" A third question is "The system has been running fine for months, but all of a sudden, it's crashing occasionally. I can't make it happen 'on demand', but it is happening regularly. How can I track this down?"

One solution to all of these problems is to track your user logins in a special file. Each time a user logs on, insert a record with the username, date, and time into a "log" file. When they exit properly, insert a second record with the same information. In addition, include a flag indicating that they logged on or off successfully. The file would have the following structure:

```
Structure for table: F:\NEB\GHC3\APPFILES\SYS_LOGS.DBF
Field  Field Name  Type       Width    Dec    Index    Collate
    1  CNAMEUSER   Character     10
    2  CSTLOG      Character     15
    3  DDATELOG    Date           8
    4  CTIMELOG    Character      8
** Total **                      43

Structural CDX file:   F:\NEB\GHC3\APPFILES\SYS_LOGS.CDX
Index tag:   CLOGORDER
Collate: Machine
Key: CNAMEUSER+DTOS(DDATELOG)+CTIMELOG
```

A typical routine to handle the logon would look like this:

```
*
* put a login record in SYS_LOGS
*
* look for the file SYS_LOGS in APPFILES directory
* and if it doesn't exist, create it (backwards compatibility)
*
if !file( m.gcAppFilesLoc + "\SYS_LOGS.DBF")
  create table m.gcAppFilesLoc + "\SYS_LOGS" ;
    ( ;
    cNameUser c(10), ;
    cStLog c(15), ;
    dDateLog d(8), ;
    cTimeLog c(8) ;
    )
   index on cNameUser + dtos(dDateLog) + cTimeLog tag cLogOrder
endif  && !file( m.gcAppFilesLoc + "\SYS_LOGS.DBF")
*
* in either case, insert the new record and then close SYS_LOGS down
*
insert into SYS_LOGS ;
  (cNameUser, cStLog, dDatelog, cTimelog) ;
  values ;
  (m.gcNameUser, "GOOD LOGIN", date(), time())
use in SYS_LOGS
*
* set the "#3" flag so that we know we got this far during our login procedure
*
m.glSetup3IsOK = .T.
```

The memory variable "m.gcNameUser" is the name of the user currently trying to log on. It is initialized to an empty string in the case of no user login (a single user application, for instance.) All the "system files" are located in the "APPFILES" directory, the name of which is indicated by the memory variable "m.gcAppFilesLoc."

And the routine to handle the logout would look like this:

```
*
* if we logged in successfully during startup, put a corresponding record in the
* LOG file indicating we're leaving successfully
*
if m.glSetup3IsOK
  insert into SYS_LOGS ;
    (cNameUser, cStLog, dDatelog, cTimelog) ;
    values ;
    (m.gcNameUser, "GOOD LOGOUT", date(), time())
  use in SYS_LOGS
endif && m.glSetup3IsOK
```

The logout procedure is fairly straightforward, but the need for the "if m.glSetup3IsOK" wrapper might not be immediately apparent. The startup procedure does a number of things, including checking the environment, setting up public memvars, looking for system files and so on, all before the login screen displays. If any of those events fails, we don't display the login screen, and so have no need to insert a "login" or a "logout" record in the LOG file.

Some people might wonder why I don't "punch the user in and out" by using a single record that would track the status and the date/time "in and out." I prefer to insert two short records for a couple of reasons. First, the space savings aren't that significant, since we still have to track the date and time "in and out" - we'd save the duplication of the username and, perhaps, the status. That's only 25 bytes per login - a trivial amount of space considering the size of applications and data files these days. Second, the coding for this technique is simple almost to the point of being trivial. If you have to find the "login" record in order to "punch out", you'll add a performance hit while looking for the record. Furthermore, the programming becomes unnecessarily complex when taking into account situations like "orphaned" records - those that have a login date and time, but no logout data - and environments such as single user systems without user login names.

Now when the user claims they've never rebooted, you can take a look at the LOG file and see how many "GOOD LOGIN" records there are without matching "GOOD LOGOUT" records. If they've sworn that they've tested the system for hours every day, you have the proof of if they've really been doing so. And in the event of those mysterious bugs, you've now got more information about who's been logged on and for how long - so you can play Sherlock Holmes or Agatha Christie with a full set of clues.