Implications of Locking in web applications

INTRODUCTION

An area of great concern to many developers is how to implement an effective locking strategy in a web application.

Because the internet operates on stateless protocols, there is no concept of a connection. If there is no connection, there is no way of telling that a user is still using the application.

This becomes an issue where an application is being web enabled that uses record locking to control 'real time' applications. On the web, a resource locked by a user may never be released.

This document is intended to illustrate the differences between locking in green screen or client server applications and locking in applications that are run over the web.

REAL TIME LOCKING AND THE WEB

In a 'real time' application a lock is placed on a record when it is selected. This lock can persist while the user completes their particular transaction, at which time the user would either save an edited version of the record or log out of the application, thus ending the session. The record lock would then be released.

However, web based applications follow a batched model, in that a user requests a page, that page is rendered and the connection broken. There is no concept of a session that may persist beyond the request and return of the page.

Also, it is entirely conceivable that a user would request a single page then abandon the application without following a 'logging off' procedure.

As such, it is impossible for a web application to implement 'real time' locking in the same way that would be possible in a green screen application.

LOCKING WEB APPLICATIONS

In order to apply a locking strategy to a web application, there are a number of approaches that may be taken.

1) When a user retrieves a record, take a copy of the information. When the user saves a modified version of the record, compare the copy with the data held in the file. If the two do not match, then the saved record has been amended since the user retrieved their information.

2) Add a timestamp of the last update of every record. When the user saves a record, compare this with the information held in the file.

3) a) When a record is retrieved, write that record to a 'Record Locked' file. When the user saves their data, remove the record from the file. Until the record is saved, anyone attempting to retrieve the record would be informed that the record is locked and would only be able to retrieve the information as 'read only'. As it is possible that the user would never actually save any information, it would also be necessary to have a timer function that continually checks when a record is locked. After a specified amount of time e.g. 10 minutes a lock would be released.

   b) This approach could be amended for those applications where editing a record may take place in more than one page so that when a user navigates to the next page in the edit sequence, the lock timestamp is updated to ensure that the lock is not released while a user is still editing a record.

4) Change the way the application works so that it does not operate in real time.

   When a change is submitted, inform them that their order request has been received and will be processed at a later time. Generally, if a customer is willing to order a product over the web, they are not likely to be expecting real time interaction with the database.

   Sites such as WWW.AMAZON.COM and WWW.JUNGLE.COM provide a service that works in this manner. They indicate items availability by stating that an item would be 'usually dispatched in 24 hrs' or 'usually dispatched in X days'.

COPY AND COMPARE

This is not a locking mechanism as such, but more of a conflict resolution strategy. The idea is that where an application might take a lock, instead it takes a copy of the data that may be modified. An example of how this might be achieved in BASIC is detailed below.

Somewhere in the initialisation section of your program, declare a common block to hold the data that the application needs to copy.

```
* Create named common for saved records
COMMON /SavedRecords/Records(10)
```

When the application would normally take a lock, save the contents away to one of the common block variables. In the example below, there are two files which might normally be locked, the Employee file, and the Department file.

```
SUB edit(EmpID,DeptID)

     * include saved records
     COMMON /SavedRecords/Records(10)

     * Open the files
     CALL sysopen("EMPLOYEE",employee,html)
     CALL sysopen("DEPARTMENT",department,html)

     * Read employee and take copy
     READ emprec FROM employee,EmpID THEN
          Records(1) = emprec
     END

     * Read department and take copy
     READ deptrec FROM department,Deptid THEN
          Records(2) = deptrec
     END

     …etc
```

As the application comes to the point where the records would be written back to the files, the saved copies are compared with the current contents of the files. If the saved values are the same as the file values, the application can safely write the new values. If the values differ, it shows that someone has modified the files since the copy was taken.

```
* include saved records
COMMON /SavedRecords/Records(10)

* Read the current values from the files
READ Curremp FROM employee,EmpID
READ Currdept FROM department,EmpID

* Have things changed ?
IF Curremp = Records(1) AND Currdept = Record(2) THEN

     * Write away the records
     WRITE emp ON employee,EmpID
     WRITE dept ON department,DeptID

END ELSE

     * Conflict resolution code goes here
END
```

If the contents of the files have changed, there are a number of things that the application can do. In the first instance, it's probably a good idea to log the fact that there's been a conflict somewhere.

After logging, there are four main choices as to what the application could do:

1. The write could continue as if nothing had happened.
2. The conflict could be presented to the user, and the user could decide whether to continue or to back out.
3. The conflict could be presented to the user, and the write prevented by the application.
4. The application could resolve the conflict by following a set of business rules.

Obviously, not all of these options will be appropriate in every case, and this approach needs to be evaluated before implementation.

TIMESTAMPED UPDATES

Again this is a conflict resolution strategy. The idea is that where an application might take a lock, instead it reads the last update timestamp from the record that is to be modified.

When the record is to be written, it checks that no updates have occurred since by checking the stored timestamp against the original. If the records haven't changed, the write proceeds as normal.

If any of the records have been updated, the conflict would be logged, and resolved in the same way as detailed in the previous section.

APPLICATION LOCKS

Writing a completely new locking mechanism is probably going to involve the most effort out of all of these alternatives, but it will give the highest degree of flexibility.

Essentially, the main problem with the built-in locking mechanism is that record locks are too absolute. A record is either locked and will remain so until released, or it is unlocked.

What is needed is a mechanism that can attach a lock to a record which doesn't remain until released, but times out. This way, a resource cannot remain locked forever, even if the application user switches their browser off.

The central part of any new locking mechanism is somewhere to store the locks. A single file can be used to store all locks for the application, or separate files may be used.

When an application would normally take a lock, it first checks the lock table to ensure that someone else hasn't locked the record. Assuming they haven't, the application writes a lock record away to the file with the following information:

Filename
Record Key
TimeStamp
Lock lifespan

This record may be deleted in one of two ways. Firstly, when the application writes away the updated record, the lock record is deleted.

Secondly, another program is required to continuously read the lock table and delete any lock records which have expired.

Obviously, the situation may arise where the lock expires before the application has had a chance to write the updated record. If this is a concern, it is a good idea to combine this strategy with one of the conflict resolution mechanisms.

If there are many pages involved in the update process, the lock may be refreshed as the user steps from page to page. In extreme cases, a periodic submit could be coded into the web builder application to refresh the lock.

APPLICATION DESIGN

If the need for record locking can be completely 'designed out' then the whole issue of locking disappears. However, achieving this is not always as easy as it sounds.

Typically if an application displays information that could be interpreted as being live, then the user will expect that data to be live. Avoiding this kind of display will go a long way to changing the users' expectations of the real-time nature of the system.

Giving the user more information about how the application will work will help with their expectations too. A message stating that all requests are dealt with within 24 hours immediately changes the users' expectations away from expecting an immediate answer.

Typically in these kinds of examples, record updates are queued up and processed in batch behind the scenes. The user of your application receives a notification about the success (or otherwise) of their request after the event.

If, having reviewed the design, the application must deliver real time information via the web application, then one of the other approaches must be adopted.

Document Created 2nd August 2001
Authors: Simon Rees & Martin Bailey