

SPECIAL OFFERS Keep up with new releases and promotions. [Sign up to hear from us.](#)

books, eBooks, and digital learning

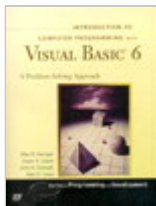
[Home](#) > [Articles](#) > [Programming](#) > [Visual Basic](#)

Preventing Multi-User Concurrency Problems

Aug 6, 2001

[Contents](#) [Print](#)[< Back](#) [Page 2 of 6](#) [Next >](#)

Like this article? We recommend

[Introduction to Computer Programming with Visual Basic 6: A Problem-Solving Approach](#)[Learn More](#)[Buy](#)

InformIT Promotional Mailings & Special Offers

I would like to receive exclusive offers and hear about products from InformIT and its family of brands. I can unsubscribe at any time.

[Privacy Notice](#)**Email Address**

Possible Solutions

There are many techniques that can be used to prevent multiuser concurrency problems. I'll present a few of the most common ones here.

Ignore It

The simplest technique is to just ignore it, hoping it will never happen; or if it does happen, that there won't be a terrible outcome. This technique may sound unacceptable, but depending on the application, it can successfully be used. Consider an application in which users are in charge of updating their own data, such as their credit card information at an online auction site. In this kind of application, it would be very rare for the lost update problem to occur because a single user should not be in two places at once. If he was and it did cause a lost update problem, you might argue that it was the user's own fault. Of course, this technique has the benefit of not requiring any additional code, but is usually unacceptable in many applications. It's especially improper for those applications in which money is involved.

Locking

Another popular technique for preventing lost update problems is to use locking techniques. With this approach, when a record is retrieved and will likely be updated (or even when there is only a remote possibility that it will be updated), the application tells the DBMS to lock the record so other processes can't retrieve and update it. It basically tells the DBMS, "I'm using this record, so don't let anyone else have it." Although this type of locking, more formally called *pessimistic locking*, can be used to prevent lost update problems, it introduces its own set of problems.

Not all DBMSs support the concept of record locking; some support only so-called *page locking*. Page locks lock a large chunk of data at a time—for example, 4KB—

so they can potentially prevent more than one record from being retrieved. Still worse, some DBMSs support locking only at the table level, and some don't support it at all.

Another problem with locking is the way the DBMS handles a request for a record that is already locked. When this occurs, the DBMS won't immediately return an error saying the record is being used. Instead, it waits for a period of time, hoping the process that locked it will free the record in question. Eventually a "time-out" error will be returned if the DBMS gives up waiting for the record to be freed. This time-out period is typically a configuration setting, and it can last from a few seconds to several minutes.

If you are a user of an application that locks records, and you request a record someone else already has locked (maybe they are keeping it locked while they talk on the phone), you have to wait until either the record is freed by the other user or until the DBMS returns a "time-out" error. In either case, you have to sit there and wait while your application appears to be doing nothing. So this locking technique often results in new *blocking* problems. It also favors the user who requests the record first and takes a long time to make changes to the record, rather than favoring the user who makes changes quickly.

Locking techniques also require that a connection to the DBMS be retained for the duration of the lock, which is not easily done by Web applications that use the stateless HTTP protocol.

Read Before Write

A technique that does work for Web applications is sometimes called *read before write*. It works like this:

When a record is retrieved that might later be updated, the contents of all the values in the record are saved as user state information somewhere. This is the "before" image of the record. When the user makes changes to the data, they are saved to a "new" image of the record in memory. Before the new version of the record is written to the database, the record is retrieved again into a third, "current," image of the record. The values of each field in the before image are compared with the values of the fields in the current image. If anything is different, the application knows that another user (or process) must have made changes to the record. It then cancels the update process, and displays a message to the user stating that the record has been updated after he or she first retrieved it, and that he or she should start the update task over again from the beginning.

I like this technique because it doesn't lead to blocking problems, and it favors the user who makes changes most quickly, instead of favoring the user who retrieves the record first—as the locking technique does. Read before write also has the benefit of being completely DBMS-independent, it even works for non-DBMS data stores, and it doesn't require additional fields to be added to the database. However, if the number of fields in the record is large, it will result in a lot of extra coding to compare the before image with the current image.


Timestamping

The technique I like to use to prevent concurrency problems, and the one I implement in this article, is called *timestamping*, in which each record in the database contains a "last modified date/time" field. This value is then updated whenever a new record is inserted and when an existing record is updated. Timestamping then works much like the read before write technique, except that instead of saving the entire record as a before image, only its last modified date/time field needs to be saved as user state information. Likewise, just before changes to a record are saved to the database, only the timestamp field is retrieved and compared with the timestamp that was saved when the record was first retrieved. At this point, timestamping works just like read before write. If the timestamp values are different, the application knows that another user has made changes to the record, so it cancels the update process and an error message is displayed to the user.

I like timestamping as a general solution because

- It is DBMS-independent.
- It works even with non-DBMS data such as random access files.

- It doesn't require the application to retain a connection with the database, so it works in stateless environments such as HTTP.
- It is simpler to code than read before write.

 [Save To Your Account](#)

[< Back](#) **Page 2** of 6 [Next >](#)