Locking

Progress® Version 9.1D SQL-92 Server
Edition 1, March 2003

Contents

Introduction

The intent of this white paper is to convey information regarding database locks as they apply to transactions in general and the more specific case of how they are implemented by the Progress® SQL-92 server. We'll begin with a general overview discussing why locks are needed and how they affect transactions. Transactions and locking are outlined in the SQL standard so no introduction would be complete without discussing the guidelines set forth here. Once we have a grasp on the general concepts of locking we'll dive into lock modes, such as table and record locks and their effect on different types of database operations. Next, the subject of timing will be introduced, when locks are obtained and when they are released. From here we'll get into lock contention and deadlocks, which are multiple operations or transactions all attempting to get locks on the same resource at the same time. And to conclude our discussion on locking we'll take a look at how we can see locks in our application so we know which transactions obtain which types of locks. Finally, this white paper describes differences in locking behavior between previous and current versions of Progress and differences in locking behavior when both 4GL and SQL92 clients are accessing the same resources.

To avoid being redundant, this white paper makes reference to Progress product documentation, which is available at www.progress.com/documentation.

Locks

Why do we lock database objects?

The answer to why we lock is simple; if we didn't there would be no consistency. Consistency provides us with successive, reliable, and uniform results without which applications such as banking and reservation systems, manufacturing, chemical, and industrial data collection and processing could not exist. Imagine a banking application where two clerks attempt to update an account balance at the same time: one credits the account and the other debits the account. While one clerk reads the account balance of $200 to credit the account $100, the other clerk has already completed the debit of $100 and updated the account balance to $100. When the first clerk finishes the credit of $100 to the balance of $200 and updates the balance to $300 it will be as if the debit never happened. Great for the customer; however the bank wouldn't be in business for long.

What objects are we locking?

What database objects get locked is not as simple to answer as why they're locked. From a user perspective, objects such as the information schema[1], user tables, and user records are locked while being accessed to maintain consistency. There are other lower level objects that require locks that are handled by the RDBMS; however, they are not visible to the user. For the purposes of this discussion we will focus on the objects that the user has visibility of and control over.

[1] Schema as defined by the Webster's dictionary reads as follows, "A diagrammatic representation; an outline or model". As applied to a Relational Database Management Systems, schema is a term used to represent the metadata; tables, fields and indexes. The term "schema" used unqualified in the context of SQL may have a different meaning depending on whether or not you're a seasoned SQL user. A SQL Catalog is a container for one or more schema, at a minimum there is the Information Schema that contains the systems base tables or metadata. There can also be one or more user defined schema contained within a SQL Catalog. For the seasoned Progress 4GL user there is only one schema, a singular object representing the metadata; tables, fields and indexes. For this discussion we well refer to this object by its correct SQL name, the Information Schema. For more information about the SQL Catalog, and the SQL Schema please refer to Melton and Simon's book entitled "Understanding the New SQL: A Complete Guide".

Transactions

Now that we know why and what we lock, let's talk a bit about when we lock. A transaction is a unit of work; there is a well-defined beginning and end to each unit of work. At the beginning of each transaction certain locks are obtained and at the end of each transaction they are released. During any given transaction, the RDBMS[2], on behalf of the user, can escalate, deescalate, and even release locks as required. We'll talk about this in more detail later when we discuss lock modes. The aforementioned is all-true in the case of a normal, successful transaction; however in the case of an abnormally terminated transaction things are handled a bit differently. When a transaction fails, for any reason, the action performed by the transaction needs to be backed out, the change undone. To accomplish this most RDBMS use what are known as "save points[3]." A save point marks the last known good point prior to the abnormal termination; typically this is the beginning of the transaction. It's the RDBMS's job to undo the changes back to the previous save point as well as ensuring the proper locks are held until the transaction is completely undone. So, as you can see, transactions that are in the process to be undone (rolled back) are still transactions nonetheless and still need locks to maintain data consistency.

Locking certain objects for the duration of a transaction ensures database consistency and isolation from other concurrent transactions, preventing the banking situation we described previously. Transactions are the basis for the **ACID** [4]properties:

- ***ATOMICITY*** guarantees that all operations within a transaction are performed or none of them are performed.
- ***CONSISTENCY*** is the concept that allows an application to define consistency points and validate the correctness of data transformations from one state to the next.
- ***ISOLATION*** guarantees that concurrent transactions have no effect on each other.
- ***DURABILITY*** guarantees that all transaction updates are preserved.

What is Defined in the 1992 SQL Standard

The 1992 SQL standard does not specify how a SQL implementation should provide for data consistency and concurrency via various locking schemes. However it does specify what the expected behavior should be for active transactions in different situations. Each behavior is identified with a specific name referred to as: "Transaction isolation level" and offers varying degrees of isolation and concurrency while accessing a database.

To clearly identify expected behaviors for each transaction isolation levels, the 1992 SQL standard starts by describing different issues occurring while accessing data in a concurrent mode. These issues are called phenomena and are permitted or prevented by each isolation level resulting in varying degrees of isolation and concurrency.

---

[2] Relational Database Management System, a type of database management system (DBMS) that stores data in the form of related tables

[3] The ability of the client application to set user defined save points is not part of the SQL92 standard however it is an extension to the core SQL99 standard. The Progress database development team is currently giving thought to implementing user defined save points in a future release.

[4] For a full explanation and discussion on the ACID database properties please refer to Jim Gray and Andreas Reuter's book entitled "Transaction Processing: Concepts and Techniques".

The three phenomena described in the 1992 SQL standard are as follows:

Dirty read — Occurs when one user is updating / inserting a record while a different user is reading it, but that work is not committed to the database.

> **Hypotheses**: We assume there is no transaction control mechanism used by the RDBMS which could mean that the RDBMS is not locking any records while a user is accessing the database.
>
> **Scenario:**

```
User 1 executes:
      INSERT INTO pub.State (state, state_name, region)
      VALUES ('AB', 'Abcdefghij', 'ABCD');
User 2 executes:
      SELECT * FROM pub.State
User 2 sees: state 'AB'
User 1 executes:
      ROLLBACK WORK
User 2 has seen data that did not really exist.
```

> **Conclusion:** To prevent this phenomenon, User 2 must guarantee that the records being accessed are not currently being accessed from a different transaction started by a different user.

Non repeatable read — Occurs when one user is repeating a read operation on the same records but has updated values.

> **Hypotheses**: We assume there is no transaction control mechanism used by the RDBMS or that the RDBMS is able to check that a record about to be retrieved is used in a transaction held by a different user. This could mean that the RDBMS is not locking any records while a user is accessing the database, or that there is a mechanism to check if a record is locked by a different user.
>
> **Scenario:**

```
User 1 executes:
      SELECT * FROM pub.State
User 2 executes:
      UPDATE pub.State
      SET state_name = 'hello world'
      WHERE state = 'AK';
      COMMIT WORK;
User 1 re-executes:
      SELECT * FROM pub.State
User 1 has now updated records in the result set.
```

> **Conclusion:** To prevent this phenomenon, the RDBMS needs to use some sort of mechanism preventing any other user from updating the records User 1 has already read. This mechanism should be used until User 1 indicates that the read operation is complete.

Phantom Read — Occurs when one user is repeating a read operation on the same records but has new records in his result set.

> **Hypotheses**: We assume there is no transaction control mechanism used by the RDBMS or that the RDBMS is able to check that a record about to be retrieved is used in a transaction held by a different user and that it holds locks on records retrieved until the user indicates that the transaction is complete. This could mean that the RDBMS is not locking any records while a user is accessing the database, or that there is a mechanism to check if a record is locked by a different user and that there is a mechanism holding locks on a record until the end of the transaction.
>
> **Scenario:**

```
User 1 executes:
      SELECT * FROM pub.State
User 2 executes:
      INSERT INTO pub.State (state, state_name, region)
      VALUES ('AB', 'Abcdefghij', 'ABCD');
      COMMIT WORK;
User 1 re-executes:
      SELECT * FROM pub.State
User 1 has new records in the result set.
```

> **Conclusion:** To prevent this phenomenon, the RDBMS needs to use some sort of mechanism preventing any other user from inserting records in the table User 1 is currently accessing. This mechanism should be used until User 1 indicates that the read operation is complete.

In order to allow or prevent each of the above phenomena, 4 isolation levels are clearly identified:

- ***READ UNCOMMITTED*** (also called "dirty read") — When this isolation level is used, a transaction can read uncommitted data that later may be rolled back. The standard requires that a transaction that uses this isolation level can only fetch data but can't update, delete, or insert data.

- ***READ COMMITTED***— With this isolation level dirty reads are not possible, but if the same row is read repeatedly during the same transaction, its contents may be changed or the entire row may be deleted by other transactions.

- ***REPEATABLE READ*** — This isolation level guarantees that a transaction can read the same row many times and it will remain intact. However, if a query with the same search criteria (the same WHERE clause) is executed more than once, each execution may return different set of rows. This may happen because other transactions are allowed to insert new rows that satisfy the search criteria or update some rows in such way that they now satisfy the search criteria.

- ***SERIALIZABLE*** — This isolation level guarantees that none of the above happens. In addition, it guarantees that transactions that use this level will be completely isolated from other transactions.

Table 1 gives a clear overview of what is described above. It identifies which phenomena are either permitted or prevented by each isolation level.

Table 1

	Dirty Read	Dirty Read	Dirty Read
Read Uncommitted	**Permitted**	**Permitted**	**Permitted**
Read Committed	**Prevented**	**Permitted**	**Permitted**
Repeatable Read	**Prevented**	**Prevented**	**Permitted**
Serializable	**Prevented**	**Prevented**	**Prevented**

Note that the isolation levels are ordered according to the phenomena they either permit or prevent —the first one (READ UNCOMMITTED) is the isolation level providing the highest level of concurrency but with the lowest level of consistency. Each subsequent level provides at least as much data consistency as the one before but will result in less concurrency.

As a general rule, the more data consistency that is provided by the isolation level used from an application, the less concurrency is allowed between this application and other applications connected to the same database.

Lock Level

In order to provide a reasonable level of concurrency, locking is performed in a hierarchy that requires several lock modes across the levels of the hierarchy. For the purpose of this discussion there are three levels in our locking hierarchy, records, tables and information schema. A record is at the lower level, a table is at the medium level, and the information schema is at the highest level.

Lock Mode

In general, lock modes are prioritized tokens in a queue that indicate what action is being taken. The intent to update a given record requires a different mode of lock than to actually update the record; likewise the intent to read a record requires a different mode of lock than to actually read the record. Lock modes are needed to facilitate concurrency and provide consistency; they indicate intent and are used to stage lock requests. Lock requests are generated on the user's behalf as a result of executing a transaction, such as a database read or write.

Progress provides 6 lock modes that are described as follows:

- ***NO-LOCK (NL)*** — you have no intentions of performing an update and accuracy of the resulting set of data is not important.
- ***INTENT SHARE (IS)*** — you intend to share-lock objects at the next lower level of granularity for this object (table). That is, you intend to get share locks on the rows of this table.
- ***INTENT EXCLUSIVE (IX)*** — you intend to exclusive-lock objects at the next lower level of granularity for this object (table). That is, you intend to get exclusive locks on the rows of this table.

- ***SHARED (S)*** — you want a share-lock on the object. Getting a share-lock on an object means that you implicitly get a share-lock on all of the objects that this object contains, i.e. all of the rows for this table.

- ***SHARED WITH INTENT EXCLUSIVE (SIX)*** — you want a share-lock on the table so no one else can modify, delete, or add rows except for you.

- ***EXCLUSIVE (X)*** — you want an exclusive-lock on the object. Getting an exclusive lock on an object means that you implicitly get an exclusive lock on all of the objects that this object contains, i.e. all of the rows for this table.

Note: Since rows (records) are the lowest level in the locking hierarchy, the intent locks (IS, IX and SIX) do not apply. At the information schema level there is currently no need to use any of these intents locks. In other words, they are only used at the table level.

Table 2 depicts the lock compatibility matrix for these different lock modes. A checkmark indicates the requested lock can be granted. A cross indicates that the requested and the granted modes are not compatible so the requested lock could not be granted on the object.

This matrix applies to lock requests made by a user different than the holder of the lock.

Table 2

Requested Mode	Granted Mode						
	Lock Mode	**None**	**IS**	**IX**	**S**	**SIX**	**X**
	None	✓	✓	✓	✓	✓	✓
	IS	✓	✓	✓	✓	✓	✖
	IX	✓	✓	✓	✖	✖	✖
	S	✓	✓	✖	✓	✖	✖
	SIX	✓	✓	✖	✖	✖	✖
	X	✓	✖	✖	✖	✖	✖

How lock levels and lock modes interact to provide the behavior described in the 1992 SQL standard

Matching the transaction isolation levels behaviors:

Table 3 describes how the Progress SQL-92 server makes use of this locking scheme implementation in order to match the desired behavior described in the 1992 SQL standard.

This information is also provided here for a better understanding of how a caller might make use of this table lock implementation; however it is not intended to dictate how it is used.

Note that the information in the table applies to the requested lock strength based on the transaction isolation level in effect for a given transaction. This table does not take into consideration lock upgrades possibly resulting in a different lock strength actually being applied.

Table 3

	Insert / Update / Delete record Operation			Fetch / Select record Operation		
Isolation Level	**Information Schema Lock**	**Table Lock**	**Record Lock**	**Information Schema Lock**	**Table Lock**	**Record Lock**
Serializable	**S**	**SIX**	**X**	**S**	**S**	**S**
Repeatable Read	**S**	**IX**	**X**	**S**	**IS**	**S**
Read Committed	**S**	**IX**	**X**	**S**	**IS**	**S**
Read Uncommitted	**S**	**NL**	**NL**	**S**	**NL**	**NL**

Notes:

- As Table 3 indicates, there are no table or record locks acquired when the transaction isolation level is Read Uncommitted. In the Read Uncommitted transaction isolation level you maximize concurrency but may also read dirty data. Depending on your application, this may be acceptable.

- The primary difference between the Read Committed and Repeatable Read transaction isolation levels is that while in Repeatable Read, individual record locks are held for the duration of the transaction. For example, if your fetch criteria include all companies in the state of Idaho, each record in the result set will remain locked until all of the records meeting the criteria have been read. In the Read Committed transaction isolation level the record locks are released once the record has been read. So, as the name indicates, you will only read committed records, which may change once your result set is complete. This is known as a phantom read. If you were to re-read the same records, it's possible that records that were once visible no longer exist. To avoid this behavior, use the Repeatable Read transaction isolation level keeping in mind that as you progress towards the Serializable transaction isolation level you reduce your application's concurrency because locks are held for a longer duration.

- In the Serializable transaction isolation level a share lock on a table is held for the duration of the transaction, preventing any other transaction from updating the table.

- Any SQL operation that modifies the information schema automatically gets upgraded to the serializable transaction isolation level regardless of the user's current transaction setting.

Now that we've discussed lock modes / levels and how they affect transactions, let's take a look at lock acquisition, that is, how and when locks are acquired.

Lock Acquisition

How and when are locks acquired? So far we've talked about locking in general, how the SQL standard interprets locks via behavior, lock modes, and database objects that get locks applied to them such as tables, records, and information schema. Knowing which objects get locked and when goes a long way towards helping you develop applications that are more robust and predictable.

Since SQL uses the transaction isolation level exclusively to determine what lock mode is applied to which objects, it's extremely important to know how this translates into object locks and lock modes. This is the only way to communicate your application's intentions to the SQL engine. As you can see from Table 3 above, the strongest locks are held when the transaction isolation level is Serializable and the weakest locks are held when the transaction isolation level is Read Uncommitted. This also translates into application concurrency—the higher the transaction isolation level, the less concurrent your application will be.

Information Schema Lock

Every operation performed by the SQL server operates inside a transaction. Operating inside a transaction provides the ability to provide for the ACID properties we expect from a database. For each transaction, a information schema share-lock is acquired at the beginning of the transaction and released at the end of the transaction. This is true regardless of whether or not the transaction is committed successfully or terminated abnormally. Acquiring the information schema share-lock protects the information schema from being altered while the transaction is active. During the life of an active connection there can be many individual transactions begun and ended depending on the operations being performed. The first transaction is begun upon connection to the SQL server and is used to read the information schema. Once the information schema has been read, the transaction is ended. Each successive operation will then begin and end a transaction requiring, at a minimum, a share-lock on the information schema. While the connection is quiet, there is no active transaction and therefore no lock held on the information schema. If an operation is being performed that will modify the information schema, an exclusive lock on the information schema will be requested. For the exclusive lock on the information schema to be granted, there can be no other active transactions in the database. Once granted, the information schema lock is upgraded from a share to an exclusive lock. While this transaction is active, no other transaction can begin because the share-lock on the information schema cannot be granted while there is an outstanding exclusive lock on the information schema. Keep in mind that the lock on the information schema is above and beyond any locks obtained on tables and records via transaction isolation level settings for data manipulation operations.

Table/Record Lock

Lock acquisition for tables and records is straightforward given Tables 2 and 3 above and the information we covered so far regarding transactions. To get a record lock of sufficient strength for the operation being performed, you must first have a table lock of sufficient strength. Regardless of the current transaction isolation level, if the application's intent is to perform an operation other than a fetch, the lock mode will be in effect strengthened for the duration of that operation. That is to say, you are not prohibited from creating or updating records based on the transaction isolation level. It is the responsibility of the SQL implementation's RDBMS to provide sufficient lock escalation when an operation is being performed that requires lock upgrades.

What happens when two transactions attempt to update an object at the same time? Let's take a look at our banking example where two clerks, C1 and C2, are updating a customer's bank account balance at the same time. To update the customer's bank account balance requires an exclusive lock on two tables; the accounts table, T1, where the customer information is located and the transaction table, T2, where debits and credits are recorded. To perform the update C1 gets an exclusive lock on T1, and C2 gets an exclusive lock on T2.

C1 now tries to lock T2, and C2 tries to lock T1. Neither can proceed because each holds a resource locked exclusively that's required to perform the update. This is called a deadlock and is handled by the RDBMS via a lock wait timeout. The rule imposed by the SQL RDBMS is that a transaction will wait for a resource to become available for five seconds at which point the application will need to retry the operation. A well-designed application could prevent this type of deadlock by requiring a lock on T1 prior to requesting a lock on T2. This example was contrived to illustrate a deadlock scenario and does not represent any kind of programming best practices.

That's it in a nutshell. Simply put, the RDBMS translates SQL transaction isolation level intended behavior into locks on information schema, tables, and records of varying strengths to give the desired results.

An example:

Assuming that you want to have the behavior associated with the **"Read Committed"** transaction isolation level (as defined by the 1992 SQL standard), locking a record requires that a lock gets placed at the information schema level to prevent any information schema modification to occur while accessing the record. It also requires that an intent share (IS) lock be obtained on a table, before locking the record itself. Thus to share-lock a row of a table requires you to get a shared intent lock on the table and a shared lock on the information schema before locking a row in the table. Similarly, before getting an exclusive lock on a row of a table, you must get a shared lock on the information schema and an exclusive intent lock on the table first.

This also means (depending on your transaction isolation level) that:

- To get a Share (S) lock on a record, the table must be locked with an Intent Share (IS) lock or stronger. If the table holds a Share (S), Shared with Intent Exclusive (SIX), or Exclusive (X) lock, then record locks need not be obtained at all for that table.

- To get an Exclusive (X) lock on a record, the table must hold an Intent Exclusive (IX) lock or stronger. If the table holds an Exclusive (X) lock then record locks need not be obtained at all for that table.

In a nutshell:

The locking protocol follows two simple rules:

- **Acquire locks from the top down.** Acquire an information schema lock and then a table lock before acquiring any record locks.

- **Release locks from the bottom up.** Release record locks before releasing table locks and at last the information schema lock.

Now that we've discussed lock modes, levels, and lock acquisition and how they affect transactions, let's take a look at lock visibility, that is, how we can see locks that have been acquired.

Lock Visibility

How can the user see what locks are in effect at any given time? There are a couple ways to accomplish this with the tools available in any Progress installation. See the *Progress Database Administration Guide and Reference* for details. PROMON is probably the most widely used utility for monitoring locks. Using

Virtual System Tables (VSTs) is also an option. Basic PROMON provides “canned” reports that you can use to get a feel for what is happening with regards to locks.

The “Record Locking Table” option displays locks that are being held at any given time. This information can be used to see the locks that are currently being held in an attempt to resolve a locking conflict. The information presented includes the ID and Name of the user holding the lock, the lock chain ID, the Record ID, the Table number and the type of lock, and a flag indicating the state of the lock. Additional information on this and other PROMON tables can be found in the *Progress Database Administration Guide and Reference*. Here is sample output:

```
Record Locking Table:
Usr Name     Chain #    Rec-id    Table Lock Flags
 44 jfj      REC  105       103        2 SHR   L
 41 jfj      REC  105       103        2 EXCL  Q H
 44 jfj      REC  269     10240        2 SHR   L
 44 jfj      REC  270     10241        2 SHR   L
 44 jfj      REC  307     10278        2 SHR   L
 44 jfj      REC  686     10657        2 SHR   L
 44 jfj      REC  707       705        2 SHR   L
 44 jfj      REC  742       740        2 SHR   L
 44 jfj      REC  771       769        2 SHR   L
 44 jfj      REC  772       770        2 SHR   L
 44 jfj      REC  774       772        2 SHR   L
 44 jfj      REC  803       801        2 SHR   L
 44 jfj      REC  836       834        2 SHR   L
 44 jfj      REC  837       835        2 SHR   L
 44 jfj      REC  867       865        2 SHR   L
 42 jfj      REC  890     20832        4 EXCL  L
 44 jfj      REC  900       898        2 SHR   L
 44 jfj      REC  903       901        2 SHR   L
 44 jfj      REC  941     10912        2 SHR   L
```

The “Locking and Waiting Statistics” option displays statistics regarding locks. The information in this report is cumulative for the life of the process. This information can give you a feel for your system’s concurrency with regards to locks. The first two lines display cumulative locking statistics. The information presented includes the Lock, whether a Lock or Wait, the User ID and Name of the user, the Record ID, the number of times a Transaction Lock was issued, and the total number of times a Information schema lock was obtained for that lock type. Here is sample information that this table provides:

```
Locking and Waiting:
Type Usr Name       Record    Trans   Schema
Lock 999 TOTAL...    89796      824        0
Wait 999 TOTAL...      142        4        0
Lock   0 jfj             0        0        0
Wait   0 jfj             0        0        0
Lock  41 jfj         39466      311        0
Wait  41 jfj            25        1        0
Lock  42 jfj         19216      228        0
Wait  42 jfj            30        0        0
Lock  43 jfj         13832      103        0
Wait  43 jfj            30        1        0
Lock  44 jfj          8579       87        0
Wait  44 jfj            19        0        0
Lock  45 jfj          5550       45        0
Wait  45 jfj            22        2        0
Lock  46 jfj             0        0        0
Wait  46 jfj             0        0        0
Lock  47 jfj          2808       23        0
Wait  47 jfj            11        0        0
Lock  48 jfj           345       27        0
Wait  48 jfj             5        0        0
```

The “Transaction Control” option displays information regarding individual transactions. The information in this table is useful for identifying when transactions are active and which user is holding them. The information presented includes the User ID and Name of the user with the transaction, the Transaction ID, the Date and Time the transaction began, the Ready to commit state, whether a transaction is Limbo, and other Coordinator information with regards to distributed transactions.. Here is sample information that this table provides:

```
Transaction Control:
Usr Name      Trans   Login    Time  R-comm?  Limbo?   Crd? Coord    Crd-task
 41 jfj          8008 09/20/02 12:47 no       no       no                   0
 42 jfj          7939 09/20/02 12:47 no       no       no                   0
 43 jfj          7556 09/20/02 12:48 no       no       no                   0
 44 jfj          7973 09/20/02 12:48 no       no       no                   0
 45 jfj          7987 09/20/02 12:48 no       no       no                   0
 47 jfj          7988 09/20/02 12:48 no       no       no                   0
 48 jfj          7650 09/20/02 12:49 no       no       no                   0
 49 jfj          7985 09/20/02 12:49 no       no       no                   0
 50 jfj          8001 09/20/02 12:49 no       no       no                   0
```

In addition to the basic PROMON displays, there are a couple of other displays under the PROMON R&D menu. You can access the R&D Menu by typing “r&d” at the main PROMON menu. There are activity, status, and other lock displays that can help you understand what is happening with regards to locking on the system.

Another option that provides information on locks is Progress Virtual System Tables (VSTs). VSTs are provided so that users can “roll their own” reports from data collected in the applicable VST. Of interest to anyone looking to understand more about locking are the following VSTs:

Lock Table Activity (_ActLock) — Displays lock-table activity, including the number of share, exclusive, upgrade, Rec Get, and redundant requests; the number of exclusive, Rec Get, share, and upgrade grants; the number of exclusive, Rec Get, share, and upgrade waits; the number of downgrades, transactions committed, cancelled requests, and database up time.

Lock Table Status File (_Lock) — Displays the status of the lock table, including the user number, the user name, lock type, record ID, number, flags, and chain.

Lock Request File (_LockReq) — Displays information about lock requests, including user name and number, record locks and waits, information schema locks and waits, and transaction locks and waits.

Record Locking Table File (_UserLock) — Displays the contents of the record locking table, such as user name, chain, number, record ID, lock type, and flags.

Other Useful Information

Based on a number of responses to customer inquiries, this section should provide additional insight into the specifics of why locking behavior for the SQL-92 server appears as it does.

Auto-Commit and the Information Schema Lock

In Progress Version 9.1D, the scope of the information schema lock for SQL92 was changed. Prior to Version 9.1D, a share-lock on the information schema was obtained upon connection and held for the duration of the connection. This meant that no information schema modifications could be performed while users other than the one performing the modification were connected to the database. This posed complications for customers using connection pools in that they would have to shut down the connection pool to perform an information schema operation. Beginning with Version 9.1D, the information schema share-lock has been moved from the scope of a connection to the scope of a transaction. No longer is the lock held for the duration of the connection.

Another aspect of the information schema lock is the auto-commit behavior. There have been a number of inquiries regarding the information schema lock being held even though there are no active clients. One way to be sure of this is to look at the Transaction Control Table using PROMON. Because of the way auto-commit works, if your client uses auto-commit they will always be inside of a transaction. With auto-commit active each time a statement is executed the previous statement is committed. This relieves the user of the task of committing every statement. So, unless you explicitly commit your transactions, or turn auto-commit off, it's likely that you will run into this situation.

Table Locks and the Progress 4GL

Prior to the introduction of the SQL92 Server and the background work started with Progress Version 9.0A, the Progress database engine did not supported table locks. Consistency and concurrency were maintained with record locks. Because SQL relies on table and record locks to carry out the intent of transaction isolation levels, table locks were implemented in the database engine. Now, both the 4GL and SQL clients encounter table locks while executing transactions. From the 4GL point of view, table locks are somewhat transparent, and based on our previous discussion regarding table and record locking we already know how they affect the SQL client.

When we described the banking example above, we mentioned locking conflicts and what happens when two operations request the same resource at the same time. This is known as a dead lock. The SQL client will wait on a resource for five seconds before giving up, at which point the operation would need to be re-tried. The five-second wait is currently hard-coded and applies to all SQL clients. The Progress database development team is currently giving thought to changing this value in a future release. For 4GL clients, there is a Lock Wait Timeout (–lkwtmo) parameter that specifies how long a client should wait for a resource. The current default value for the Lock Wait Timeout parameter is thirty minutes. Given that a SQL client can only wait five seconds and a 4GL client could wait as long as 30 minutes, if a SQL client has a lock on a table for which a 4GL client also has requested a lock, the SQL client will timeout and give up waiting long before the 4GL client.

The reason for providing this information is to make our customers aware that there are behavioral differences when both 4GL and SQL clients are active on the same database. Knowing that the Lock Wait Timeout values between the 4GL and SQL are different and that the 4GL is subject to table locks when SQL clients are active may help explain unfamiliar or unexpected locking behavior.

Worldwide and North American Headquarters

Progress Software Corporation, 14 Oak Park, Bedford, MA 01730 USA Tel: 781 280 4000 Fax: 781 280 4095

Europe/Middle East/Africa Headquarters

Progress Software Europe B.V. Schorpioenstraat 67 3067 GG Rotterdam, The Netherlands Tel: 31 10 286 5700 Fax: 31 10 286 5777

Latin American Headquarters

Progress Software Corporation, 2255 Glades Road, One Boca Place, Suite 300 E, Boca Raton, FL 33431 USA Tel: 561 998 2244 Fax: 561 998 1573

Asia/Pacific Headquarters

Progress Software Pty. Ltd., 1911 Malvern Road, Malvern East, 3145, Australia Tel: 61 39 885 0544 Fax: 61 39 885 9473

www.progress.com

1.8 Database Interaction and Cursors

PL/SQL is tightly integrated with the underlying SQL layer of the Oracle database. You can execute SQL statements (UPDATE, INSERT, DELETE, and SELECT) directly in PL/SQL programs. You can also execute Data Definition Language (DDL) statements through the use of dynamic SQL (DBMS_SQL in Oracle7 and Oracle8, native dynamic SQL in Oracle8*i*). In addition, you can manage transactions with COMMIT, ROLLBACK, and other Data Control Language (DCL) statements.

1.8.1 Transaction Management

The Oracle RDBMS provides a transaction model based on a unit of work. The PL/SQL language supports most, but not all, of the database model for transactions (you cannot, for example, ROLLBACK FORCE). Transactions begin with the first change to data and end with either a COMMIT or ROLLBACK. Transactions are independent of PL/SQL blocks. Transactions can span multiple PL/SQL blocks, or there can be multiple transactions in a single PL/SQL block. The PL/SQL supported transaction statements are: COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION, and LOCK TABLE. Each is detailed here:

1.8.1.1 COMMIT

```
COMMIT [WORK] [COMMENT text];
```

COMMIT makes the database changes permanent and visible to other database sessions. The WORK keyword is optional and only aids readability -- it is rarely used. The COMMENT text is optional and can be up to 50 characters in length. It is only germane to in-doubt distributed (two-phase commit) transactions. The database statement COMMIT FORCE for distributed transactions is not supported in PL/SQL.

1.8.1.2 ROLLBACK

```
ROLLBACK [WORK] [TO [SAVEPOINT] savepoint_name];
```

ROLLBACK undoes the changes made in the current transaction either to the beginning of the transaction or to a *savepoint*. A savepoint is a named processing point in a transaction, created with the SAVEPOINT statement. Rolling back to a savepoint is a partial rollback of a transaction, wiping out all changes (and savepoints) that occurred later than the named savepoint.

1.8.1.3 SAVEPOINT

```
SAVEPOINT savepoint_name;
```

SAVEPOINT establishes a savepoint in the current transaction. *savepoint_name* is an undeclared identifier -- you do not declare it. More than one savepoint can be established

within a transaction. If you reuse a savepoint name, that savepoint is *moved* to the later position and you will not be able to rollback to the initial savepoint position.

1.8.1.4 SET TRANSACTION

```
SET TRANSACTION READ ONLY;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION USE ROLLBACK SEGMENT rbseg_name;
```

SET TRANSACTION has three transaction control functions:

READ ONLY

> Marks the beginning of a read-only transaction. This indicates to the RDBMS that a read-consistent view of the database is to be enforced for the transaction (the default is for the statement). This read-consistent view means that only changes committed before the transaction begins are visible for the duration of the transaction. The transaction is ended with either a COMMIT or ROLLBACK. Only LOCK TABLE, SELECT, SELECT INTO, OPEN, FETCH, CLOSE, COMMIT, or ROLLBACK statements are permitted during a read-only transaction. Issuing other statements, such as INSERT or UPDATE, in a read-only transaction results in an ORA-1456 error.

ISOLATION LEVEL SERIALIZABLE

> Similar to a READ ONLY transaction in that transaction-level read consistency is enforced instead of the default statement-level read consistency. Serializable transactions do allow changes to data, however.

USE ROLLBACK SEGMENT

> Tells the RDBMS to use the specifically named rollback segment *rbseg_name*. This statement is useful when only one rollback segment is large and a program knows that it needs to use the large rollback segment, such as during a month-end close operation. For example, if we know our large rollback segment is named *rbs_large*, we can tell the database to use it by issuing the following statement before our first change to data:

```
SET TRANSACTION USE ROLLBACK SEGMENT rbs_large;
```

1.8.1.5 LOCK TABLE

```
LOCK TABLE table_list IN lock_mode MODE [NOWAIT];
```

This statement bypasses the implicit database row-level locks by explicitly locking one or more tables in the specified mode. The *table_list* is a comma-delimited list of tables. The *lock_mode* is one of ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE, SHARE,

SHARE ROW EXCLUSIVE, or EXCLUSIVE. The NOWAIT keyword specifies that the RDBMS should not wait for a lock to be released. If there is a lock when NOWAIT is specified, the RDBMS raises the exception "ORA-00054: resource busy and acquire with NOWAIT specified". The default RDBMS locking behavior is to wait indefinitely.

1.8.2 Native Dynamic SQL (Oracle8i)

Native dynamic SQL introduces a new PL/SQL statement, EXECUTE IMMEDIATE, and new semantics for the OPEN FOR, FETCH, and CLOSE statement family. The former applies to single-row queries and DDL, while the latter supports dynamic multi-row queries. The syntax for these statements is:

```
EXECUTE IMMEDIATE SQL_statement_string
[INTO { define_variable_list | record |
   object_variable }]
[USING [IN | OUT | IN OUT] bind_argument_list];

OPEN cursor_variable FOR
   SELECT_statement_string;

FETCH cursor_variable INTO {define_variable_list
    | record | object_variable};
CLOSE cursor_variable;
```

The EXECUTE IMMEDIATE statement parses and executes the SQL statement in a single step. It can be used for any SQL statement except a multi-row query. *define_variable_list* is a comma-delimited list of variable names; the *bind_argument_list* is a comma-delimited list of bind arguments. The parameter mode is optional and defaults to IN. Do not place a trailing semicolon in the *SQL_statement_string*.

This is the statement that can be used to execute DDL without the DBMS_SQL package. For example:

```
EXECUTE IMMEDIATE 'TRUNCATE TABLE foo';
EXECUTE IMMEDIATE 'GRANT SELECT ON '|| tabname_v ||
   ' TO ' || grantee_list;
```

The OPEN FOR statement assigns a multi-row query to a weakly typed cursor variable. The rows are then FETCHed and the cursor CLOSEd.

```
DECLARE
   TYPE cv_typ IS REF CURSOR;
   cv cv_typ;
   laccount_no NUMBER;
   lbalance NUMBER;
BEGIN
   OPEN cv FOR
      'SELECT account_no, balance
         FROM accounts
        WHERE balance < 500';
   LOOP
```

```
      FETCH cv INTO laccount_no, lbalance;
      EXIT WHEN cv%NOTFOUND;
      -- Process the row.
   END LOOP;
   CLOSE cv;
END;
```

1.8.3 Autonomous Transactions (Oracle8i)

Autonomous transactions execute within a block of code as separate transactions from the outer (main) transaction. Changes can be committed or rolled back in an autonomous transaction without committing or rolling back the main transaction. Changes committed in an autonomous transaction are visible to the main transaction, even though they occur after the start of the main transaction. Changes committed in an autonomous transaction are visible to other transactions as well. The RDBMS suspends the main transaction while the autonomous transaction executes:

```
PROCEDURE main IS
BEGIN
   UPDATE ...-- Main transaction begins here.
   DELETE ...
   at_proc;  -- Call the autonomous transaction.
   SELECT ...
   INSERT ...
   COMMIT;   -- Main transaction ends here.
END;

PROCEDURE at_proc IS
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN        -- Main transaction suspends here.
   SELECT ...
   INSERT ...-- Autonomous transaction begins here.
   UPDATE ...
   DELETE ...
   COMMIT;   -- Autonomous transaction ends here.
END;         -- Main transaction resumes here.
```

So, changes made in the main transaction are not visible to the autonomous transaction and if the main transaction holds any locks that the autonomous transaction waits for, a deadlock occurs. Using the NOWAIT option on UPDATE statements in autonomous transactions can help to minimize this kind of deadlock. Functions and procedures (local program, standalone, or packaged), database triggers, top-level anonymous PL/SQL blocks, and object methods can be declared autonomous via the compiler directive PRAGMA AUTONOMOUS_TRANSACTION.

In the example below, the COMMIT does not make permanent pending changes in the calling program. Any rollback in the calling program would also have no effect on the changes committed in this autonomous procedure:

```
CREATE OR REPLACE PROCEDURE add_company (
   name_in   company.name%TYPE
```

```
   )
IS
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   determine_credit(name);
   create_account(name);
   ...
   COMMIT;  -- Only commit this procedure's changes.
END add_company;
```

Chapter 4: Database Locking, Scanning, And Processing

1. Overview

Before describing particular components of the IOC software, it is helpful to give an overview of three closely related topics: Database locking, scanning, and processing. Locking is done to prevent two different tasks from simultaneously modifying related database records. Database scanning is the mechanism for deciding when records should be processed. The basics of record processing involves obtaining the current value of input fields and outputting the current value of output fields. As records become more complex so does the record processing.

One powerful feature of the DATABASE is that records can contain links to other records. This feature also causes considerable complication. Thus, before discussing locking, scanning, and processing, database links are described.

2. Database Links

A database record may contain links to other records. Each link is one of the following types:

- **INLINK**: Input link, used to fetch data.
- **OUTLINK**: Output link, used to write data.

INLINKs and OUTLINKs can be one of the following: constant, database link, channel access link, or a reference to a hardware signal.

- **FWDLINK**: A forward link refers to a record that should be processed whenever the record containing the forward link completes processing.

NOTE: If a forward link is not a database link it is just ignored.

This chapter only discusses database links. Links are defined in file link.h.

Database links are referenced by calling one of the following routines:

- **dbGetLink**: The value of the field referenced by the input link retrieved.
- **dbPutLink**: The value of the field referenced by the output link is changed.
- **dbScanPassive**: The record referred to by the forward link is processed if it is passive.

A forward link only makes sense if it refers to a passive record that the application developer wants processed after the record containing the link. For input and output links, however, two other attributes can be specified by the application developer, process passive and maximize severity.

Process Passive

Process passive (PP or NPP), is either TRUE or FALSE. It determines if the linked record should be processed before getting a value from an input link or after writing a value to an output link. The linked record will be processed, via a call to dbProcess, only if the record is a passive record and process_passive is TRUE.

Maximize Severity

Maximize severity (MS or NMS), is TRUE or FALSE. It determines if alarm severity is propagated across links. For input links the alarm severity of the record referred to by the link is propagated to the record containing the link. For output links the alarm severity of the record containing the link is propagated to the record referred to by the link. In either case, if the severity is changed, the alarm status is set to LINK_ALARM.

The method of determining if the alarm status and severity should be changed is called "maximize severity". In addition to its actual status and severity, each record also has a new status and severity. The new status and severity are initially 0, which means NO_ALARM. Every time a software component wants to modify the status and severity, it first checks the new severity and only makes a change if the severity it wants to set is greater than the current new severity. If it does make a change, it changes the new status and new severity, not the current status and severity. When database monitors are checked, which is normally done by a record processing routine, the current status and severity are set equal to the new values and the new values reset to zero. The end result is that the current alarm status and severity reflect the highest severity outstanding alarm. If multiple alarms of the same severity are present the status reflects the first one detected.

3. Database Locking

The purpose of database locking is to prevent a record from being processed simultaneously by two different tasks. In addition, it prevents "outside" tasks from changing any field while the record is being processed.

The following routines are provided for database locking.

```
dbScanLock(precord);
dbScanUnlock(precord);
```

The basic idea is to call dbScanLock before performing any operations that can modify database records and calling dbScanUnlock after the modifications are complete. Because of database links (Input, Output, and Forward) a modification to one record can cause modification to other records. All records linked together, except possibly for input links

declared NPP and NMS, are placed in the same lock set. dbScanLock locks the entire lock set not just the record requested. dbScanUnlock unlocks the entire set.

The following rules determine when the lock routines must be called:

1. The periodic, I/O event, and event tasks lock before and unlock after processing:
2. dbPutField locks before modifying a record and unlocks afterwards.
3. dbGetField locks before reading and unlocks afterwards.
4. Any asynchronous record support completion routine must lock before modifying a record and unlock afterwards.

All records linked via OUTLINKs and FWDLINKs are placed in the same lock set. Records linked via INLINKs with process_passive or maximize_severity TRUE are also forced to be in the same lock set. The lock sets are determined during IOC initialization.

4. Database Scanning

Database scanning is the mechanism that requests a database record be processed. Four types of scanning are possible:

1. **Periodic** - Records are scanned at regular intervals.
2. **I/O event** - A record is scanned as the result of an I/O interrupt.
3. **Event** - A record is scanned as the result of any task issuing a post_event request.
4. **Passive** - A record is scanned as a result of a call to dbScanPassive. dbScanPassive will issue a record processing request if and only if the record is passive and is not already being processed.

A dbScanPassive request results from a task calling one of the following routines:

- **dbScanPassive**: Only record processing routines, dbGetLink, dbPutLink, and dbPutField call dbScanPassive. Record processing routines call it for each forward link in the record.
- **dbPutField**: This routine changes the specified field and then, if the field has been declared process_passive, calls dbScanPassive. Each field of each record type has the attribute process_passive declared TRUE or FALSE in the ASCII definition file. This attribute is a global property, i.e. the application developer has no control of it. This use of process_passive is used only by dbPutField. If dbPutField finds the record already active (this can happen to asynchronous records) and it is supposed to cause it to process, it arranges for it to be processed again, when the current processing completes.
- **dbGetLink**: If the link specifies process passive, this routine calls dbScanPassive. Whether or not dbScanPassive is called, it then obtains the specified value.
- **dbPutLink**: This routine changes the specified field. Then, if the link specifies process passive, it calls dbScanPassive. dbPutLink is only called from record processing routines. Note that this usage of process_passive is under the control of the application developer. If dbPutLink finds the record already active because

of a dbPutField directed to this record then it arranges for the record to be processed again, when the current processing completes.

All non-record processing tasks (Channel Access, Sequence Programs, etc.) call dbGetField to obtain database values. dbGetField just reads values without asking that a record be processed.

5. Record Processing

A record is processed as a result of a call to dbProcess. Each record support module must supply a routine process. This routine does most of the work related to record processing. Since the details of record processing are record type specific this topic is discussed in greater detail in "Record And Device Support" on page 65.

6. Guidelines for Creating Database Links

The ability to link records together is an extremely powerful feature of the IOC software. In order to use links properly it is important that the Application Developer understand how they are processed. As an introduction consider the following example (Figure 4-1):

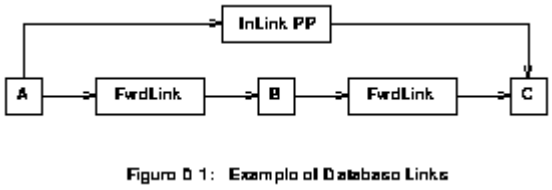


Figure 4-1: Example of Database Links

Assume that A, B, and C are all passive records. The notation states that A has a forward link to B and B to C. C has an input link obtaining a value from A. Assume, for some reason, A gets processed. The following sequence of events occurs:

1. A begins processing. While processing a request is made to process B.
2. B starts processing. While processing a request is made to process C.
3. C starts processing. One of the first steps is to get a value from A via the input link.
4. At this point a question occurs. Note that the input link specifies process passive (signified by the PP after InLink). But process passive states that A should be processed before the value is retrieved. Are we in an infinite loop? The answer is no. Every record contains a field pact (processing active), which is set TRUE when record processing begins and is not set FALSE until all processing completes. When C is processed A still has pact TRUE and will not be processed again.
5. C obtains the value from A and completes its processing. Control returns to B.
6. B completes returning control to A
7. A completes processing.

This brief example demonstrates that database links needs more discussion.

Rules Relating to Database Links

Processing Order

The processing order is guaranteed to follow the following rules:

1. Forward links are processed in order from left to right and top to bottom. For example the following records are processed in the order FLNK1, FLNK2, FLKN3, FLNK4 (Figure 4-2).

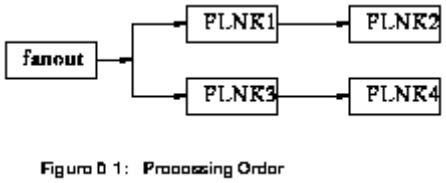


Figure 4-2: Processing Order

2. If a record has multiple input links (calculation and select records) the input is obtained in the natural order. For example if the fields are named INPA, INPB, ..., INPL, then the links are read in the order A then B then C, etc. Thus if obtaining an input results in a record being processed, the processing order is guaranteed.
3. All input and output links are processed before the forward link.

Lock Sets

All records, except possibly for NPP & NMS input links, linked together directly or indirectly are placed in the same lock set. When dbScanLock is called the entire set, not just the specified record, is locked. This prevents two different tasks from simultaneously modifying records in the same lock set.

PACT - processing active

Each record contains a field pact. This field is set TRUE at the beginning of record processing and is not set FALSE until the record is completely processed. In particular no links are processed with pact FALSE. This prevents infinite processing loops. The example given at the beginning of this chapter gives an example. It will be seen in Section 7 on page 32 and Section 8 on page 33 that pact has other uses.

Process Passive: Link option

Input and output links have an option called process passive. For each such link the application developer can specify process passive TRUE (PP) or process passive FALSE (NPP). Consider the following example (Figure 4-3):

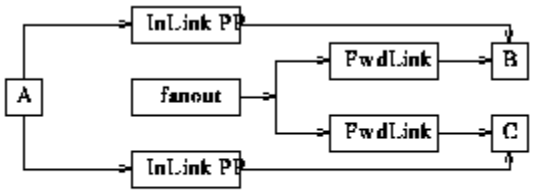


Figure 0 1: Incorrect Link Definition

Figure 4-3: Incorrect Link Definition

Assume that all records except fanout are passive. When the fanout record is processed the following sequence of events occur:

1. Fanout starts processing and asks that B be processed.
2. B begins processing. It calls dbGetLink to obtain data from A.
3. Because the input link has process passive true, a request is made to process A.
4. A is processed, the data value fetched, and control is returned to B
5. B completes processing and control is returned to fanout. Fanout asks that C be processed.
6. C begins processing. It calls dbGetLink to obtain data from A.
7. Because the input link has process passive TRUE, a request is made to process A.
8. A is processed, the data value fetched, and control is returned to C.
9. C completes processing and returns to fanout
10. The fanout completes

Note that A got processed twice. This is unnecessary. If the input link to C is declared no process passive then A will only be processed once. Thus we should have (Figure 4-4).

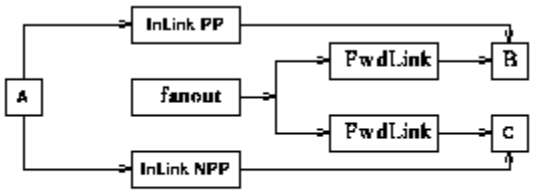


Figure 0 1: Correct Link definition

Figure 4-4: Correct Link definition

Process Passive: Field attribute

Each field of each database record type has an attribute called process_passive. This attribute is specified in the ASCII record definition file. It is not under the control of the application developer. This attribute is used only by dbPutField. It determines if a passive record will be processed after dbPutField changes a field in the record. Consult the record specific information in the record reference manual for the setting of individual fields.

Maximize Severity: Link option

Input and output links have an option called maximize severity. For each such link the application developer can specify maximize severity TRUE (MS) or maximize severity FALSE (NMS).

When database input or output links are defined via DCT, the application developer can specify if alarm severities should be propagated across links. For input links the severity is propagated from the record referred to by the link to the record containing the link. For output links the severity of the record containing the link is propagated to the record referenced by the link. The alarm severity is transferred only if the new severity will be greater than the current severity. If the severity is propagated the alarm status is set equal to LINK_ALARM. See `Maximize Severity" on page 28 for details.

7. Guidelines for Synchronous Records

A synchronous record is a record that can be completely processed without waiting. Thus the application developer never needs to consider the possibility of delays when he defines a set of related records. The only consideration is deciding when records should be processed and in what order a set of records should be processed.

Lets review the methods available to the application programmer for deciding when to process a record and for enforcing the order of record processing.

1. A record can be scanned periodically (at one of several rates), via I/O event, or via Event.
2. For each periodic group and for each Event group the phase field can be used to specify processing order.
3. The application programmer has no control over the record processing order of records in different groups.
4. The disable fields (SDIS, DISA, and DISV) can be used to disable records from being processed. By letting the SDIS field of an entire set of records refer to the same input record, the entire set can be enabled or disabled simultaneously. See the Record Reference Manual for details.
5. A record (periodic or other) can be the root of a set of passive records that will all be processed whenever the root record is processed. The set is formed by input, output, and forward links.
6. The process_passive option specified for each field of each record determines if a passive record is processed when a dbPutField is directed to the field. The application developer must be aware of the possibility of record processing being triggered by external sources if dbPutFields are directed to fields that have process_passive TRUE.
7. The process_passive option for input and output links provides the application developer control over how a set of records are scanned.
8. General link structures can be defined. The application programmer should be wary, however, of defining arbitrary structures without carefully analyzing the processing order.

8. Guidelines for Asynchronous Records

The previous discussion does not allow for asynchronous records. An example is a GPIB input record. When the record is processed the GPIB request is started and the processing routine returns. Processing, however, is not really complete until the GPIB request completes. This is handled via an asynchronous completion routine. Lets state a few attributes of asynchronous record processing.

During the initial processing for all asynchronous records the following is done:

1. pact is set TRUE
2. Data is obtained for all input links
3. Record processing is started
4. The record processing routine returns

The asynchronous completion routine performs the following algorithm:

1. Record processing continues
2. Record specific alarm conditions are checked
3. Monitors are raised
4. Forward links are processed
5. pact is set FALSE.

Lets note a few attributes of the above rules:

1. Asynchronous record processing does not delay the scanners.
2. Between the time record processing begins and the asynchronous completion routine completes, no attempt will be made to again process the record. This is because pact is TRUE. The routine dbProcess checks pact and does not call the record processing routine if it is TRUE. Note, however, that if dbProcess finds the record active 10 times in succession, it raises a SCAN_ALARM.
3. Forward and output links are triggered only when the asynchronous completion routine completes record processing.

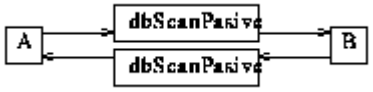
With these rules the following works just fine:

When dbProcess is called for record ASYN, processing will be started but dbScanPassive will not be called. Until the asynchronous completion routine executes any additional attempts to process ASYN are ignored. When the asynchronous callback is invoked the dbScanPassive is performed.

Problems still remain. A few examples are:

Infinite Loop

Infinite processing loops are possible.



Assume both A and B are asynchronous passive records and a request is made to process A. The following sequence of events occur.

1. A starts record processing and returns leaving pact TRUE.
2. Sometime later the record completion for A occurs. During record completion a request is made to process B. B starts processing and control returns to A which completes leaving its pact field TRUE.
3. Sometime later the record completion for B occurs. During record completion a request is made to process A. A starts processing and control returns to B which completes leaving its pact field TRUE.

Thus an infinite loop of record processing has been set up. It is up to the application developer to prevent such loops.

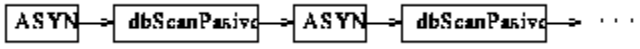
Obtain Old Data

A dbGetLink to a passive asynchronous record can get old data.

If A is a passive asynchronous record then the dbGetLink request forces dbProcess to be called for A. dbProcess starts the processing and returns. dbGetLink then reads the desired value which is still old because processing will only be completed at a later time.

Delays

Consider the following:



The second ASYN record will not begin processing until the first completes, etc. This is not really a problem except that the application developer must be aware of delays caused by asynchronous records. Again, note that scanners are not delayed, only records downstream of asynchronous records.

Task Abort

If the processing task aborts and the watch dog task cleans up before the asynchronous processing routine completes what happens? If the asynchronous routine completes before the watch dog task runs everything is okay. If it doesn't? This is a more general question of the consequences of having the watchdog timer restart a scan task. EPICS currently does not allow scanners to be automatically restarted.

9. Cached Puts

The rules followed by dbPutLink and dbPutField provide for "cached" puts. This is necessary because of asynchronous records. Two cases arise.

The first results from a dbPutField, which is a put coming from outside the database, i.e. Channel Access puts. If this is directed to a record that already has pact TRUE because the record started processing but asynchronous completion has not yet occurred, then a value is written to the record but nothing will be done with the value until the record is again processed. In order to make this happen dbPutField arranges to have the record reprocessed when the record finally completes processing.

The second case results from dbPutLink finding a record already active because of a dbPutField directed to the record. In this case dbPutLink arranges to have the record reprocessed when the record finally completes processing. Note that it could already be active because it appears twice in a chain of record processing. In this case it is not reprocessed because the chain of record processing would constitute an infinite loop.

Note that the term caching not queuing is used. If multiple requests are directed to a record while it is active, each new value is placed in the record but it will still only be processed once, i.e. last value wins.