Other concurrency systems:

Time Stamping.

A time stamp is a unique number given to each Tx, usually the time at which the 
Tx started, i.e. before its first instruction (via the system clock).
The time stamp for Tx Ti is denoted TS(Ti).

read_TS(X) returns the time stamp equal to the largest (latest) TS of all Tx's that 
have successfully read X.
write_TS(X) returns the TS equal to the largest (latest) TS of all Tx's that have 
successfully written X

The protocol
1. If a Tx T issues write(X).
   a. if read_TS(X) > TS(T), then abort and rolled back T. (Some Tx that 
      started after T has read X, so T cannot be allowed to write it )
   b. if write_TS(X) > TS(T), then abort and rolled back T.
   c. otherwise execute write(X) and set write_TS(X) = TS(T)
2. If T issues read(X) and 
   a. if write_TS(X) > TS(T) then abort and rolled back T (A Tx that started 
      after T has written X..T has lost its chance)
   b. otherwise execute read(X) and set read_TS(X) = max(read_TS(X), 
      TS(T))

When a Tx is aborted and rolled back, the system assigns it a new timestamp when it is 
restarted.

Example 1:
Consider the following transactions T1 and T2
T1: Read(B), Read(A), A := A – 25, Write(A), display (A+B)
T2: Read(B), B := B – 50, Write(B), Read(A), A := A + 50; Write(A), display (A+B)

And this is their sequential execution, where the TS(T1) < TS(T2):

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>B := B – 50</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
</tr>
<tr>
<td></td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td>A := A + 50</td>
</tr>
<tr>
<td></td>
<td>Write(A)</td>
</tr>
<tr>
<td>Read(A)</td>
<td>Display(A + B)</td>
</tr>
<tr>
<td>Display(A + B)</td>
<td>Display(A + B)</td>
</tr>
</tbody>
</table>
This is an example of a Read-Too-Late. Here is clear that T1 should be aborted and rolled back, because when T1 tries to execute Read(A), the following inequality holds: TS(T1) < write_TS(A)

Example 2:
Consider now the following execution of the same two Txs, but now TS(T1) > TS(T2).

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(B)</td>
<td>Write(B)</td>
</tr>
<tr>
<td>B := B - 50</td>
<td>Read(A)</td>
</tr>
<tr>
<td>A := A + 50</td>
<td>Write(A)</td>
</tr>
<tr>
<td>Display(A + B)</td>
<td>Display(A + B)</td>
</tr>
</tbody>
</table>

This is an example of a Write-Too-Late. Here is clear that T2 should be aborted and rolled back, because when it tries to execute Write(B), the following inequality holds: TS(T2) < read_TS(B). The value of B that T2 tries to write is obsolete (a younger transaction, T1, has read the value)

Example 3:
Consider now the following two transactions, where T2 should be executed after T1:

T1: Read(C), C := C - 25, Write(C), Read(B), B := B + 25, Write(B), display (C+B)
T2: B := 50, Write(B), Read(A), A := A + 50, Write(A), display (A+B)

And this is the sequential execution, where the TS(T1) < TS(T2):

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(C)</td>
<td>B := 50</td>
</tr>
<tr>
<td>C := C - 25</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Write(C)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>A := A + 50</td>
</tr>
<tr>
<td>B := B + 25</td>
<td>Write(A)</td>
</tr>
</tbody>
</table>
Display(C + B) 

Display(A + B)

This is another example of Write-Too-Late. Here is clear that T1 should be aborted and rolled back, because when it tries to execute Write(B), the following inequality holds: \( TS(T1) < write_{TS}(B) \).

The value of B that T1 tries to write is obsolete (a younger transaction, T2, has written the value). Note that in this case \( read_{TS}(B) = TS(T1) \).

The timestamp protocol can be modified to allow greater potential concurrency. Consider example 3. We said that when T1 tries to Write(B), \( TS(T1) < write_{TS}(B) \), and therefore according to the protocol T1 should be rolled back. But this rollback is unnecessary. Since T2 has already written B, the value that T1 is attempting to write is one that will never be read. No transaction Ti that should have read the value of B written by T1 could read the value written by T2, because when Ti tried to read B, it would have been aborted and rolled back, because \( TS(Ti) < write_{TS}(B) \) (Read-too-late). Future reads of B will want to value of B written by T2 or future Txs. This idea, that writes can be skipped when a write with a later write_TS is already in place, is called the **Thomas write rule**, and can be formulated in the new protocol, where only the point 1.b is different from the previous one.

1. If a Tx T issues write(X),
   a. if \( read_{TS}(X) > TS(T) \), then abort and rolled back T. (The value of X that T is producing was previously needed, and it had been assumed that the value would never be produce).
   b. if \( write_{TS}(X) > TS(T) \), then T is attempting to write an obsolete value of X. This write(X) is ignored.
   c. otherwise execute write(X) and set \( write_{TS}(X) = TS(T) \)

2. If T issues read(X) and
   a. if \( write_{TS}(X) > TS(T) \) then abort and rolled back T (A Tx that started after T has written X..(.T has lost its chance)
   b. otherwise execute read(X) and set \( read_{TS}(X) = \max(read_{TS}(X), TS(T)) \)

This difference between the two protocols is that whenever \( TS(T) \geq read_{TS}(X) \), if T issues a write(X), and \( TS(T) < write_{TS}(T) \), this obsolete write can be ignored.

Time stamping has some serious drawbacks:

If Tx T has to be aborted and rolled back, then any Tx that has read a value written by T must also be rolled back (cascading rolled back).

Livelock can occur. A Tx can wind up constantly be started and aborted. (The cyclic restart problem)

Time stamping also has some serious benefits. If T writes X then any Tx that starts after T can read X..no locking
Time stamping is often a better choice for concurrency control in applications with long Tx time, like CAD or CAM...this is especially true in systems (like many OO databases) that allow multiple versions of a data item...to wit:

Multi-version concurrency control

Each data item is allowed multiple versions: X0, X1,... For data item, the DBMS keeps track of version numbers and for each version, two time stamps are kept:

read_TS(Xi) and write_TS(Xi)

read_TS(Xi) is the largest TS of all Tx's that have read Xi.
write_TS(Xi) is the TS of the Tx whose write(X) caused version i to be created.

Preliminary note, in what follows, when Ti executes write(X) on version Xk a new version of X, Xk+1 is created and write_TS(Xk+1) = TS(Ti) and when Ti executes read(X) on version Xk, then read_TS(Xk) = max(read_TS(Xk), TS(Ti)).

The protocol.

1. If Tj issues write(X). Let t = max{write_TS(Xk) for all versions Xk} and let Xi be the version with write_TS(Xi) being t.
   If t <= TS(Tj) and TS(Tj) < read_TS(Xi), then abort and rolled back Tj.
   Else create a new version Xn of X and set write_TS(Xn) = TS(Tj) and read_TS(Xn) = TS(Tj)
   /* in English..a "newer" Tx has read an older version of X so Tj can't be allowed to write X */

2. If Tj issues read(X). Let t be the largest write_TS(Xk) that is still smaller than TS(Tj). Let Xi be the version whose TS that is (ie t = write_TS(Xi) ) then execute read(X) on Xi and set read_TS(Xi) = max { read_TS(Xi), TS(Tj)}

Example: 3 Tx's are running T0, T1, T2. Assume TS(Ti) is i and that initially only one version of X exists, X0 and that read_TS(X0) = write_TS(X0) = -1.

The schedule

```plaintext
Read(T0,X) 1.
Write(T0,X) 2.
Read(T1,X) 3.
Read(T2,X) 4.
Write(T1,X) 5.
Write(T2,X) 6.
```
As with previous time stamping schemes, cascading rolled back is a major problem.

Optimistic Concurrency Control.

The idea: A "shadow" or local copy of data items is kept by a Tx. All writes are applied to this local copy with no concurrency checking. After a Tx has completed, a check is made to see if serializability has been violated. If not, the changes are applied to the actual database, else the Tx is restarted.

Each Tx must go through three phases:

1. A read phase...data items can be read from the database; however, updates are only applied to the local copies (versions) of the data items kept in the transaction workspace.

2. A validation phase...the Tx checks to see if serializability has been violated (usually using Time Stamps).

3. A write phase...if the Tx is validated, then the actual writes take place.

We need two new ideas to implement this scheme.

Write_set( Ti ) = { all items, X, written by Ti }
Read_set( Ti ) = { all items, X, read by Ti }

During the validation phase, Tj must check that any Tx Ti that has either committed or is in its validation phase satisfies one of the following conditions:

1. Ti completed its write phase before Tj began its read phase.

2. Tj starts its write phase after Ti completes its write phase AND
   read_set(Tj ) ∩ write_set (Ti ) = ∅

3. Ti completes its read phase before Tj completes its read phase AND
   read_set( Ti ) ∩ write_set(Tj ) = ∅ AND
   write_set( Tj ) ∩ write_set( Ti ) = ∅
Optimistic Concurrency Control works very well in situations where the likelihood of two Tx's competing for the same data item is low. The danger of this technique is that some Tx's have to restarted over and over again and livelock can occur.

Locking Granularity. Refers to the kind of data item that is locked:

<table>
<thead>
<tr>
<th>Fine</th>
<th>record</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td></td>
</tr>
<tr>
<td>page/block</td>
<td></td>
</tr>
<tr>
<td>table/file</td>
<td></td>
</tr>
<tr>
<td>Coarse</td>
<td>database</td>
</tr>
</tbody>
</table>

The finer the granularity, the better the concurrency but the more expensive the cost of maintaining the locking system.

Recovery Revisited. Recovery from failure is more complicated in a multi-Tx system.

1. With deferred update: Since a Tx doesn't actually make any changes to the database until sometime after its commit record is written to the log, there is no chance of concurrency problems but the Tx must hold all of its locks until the Tx actually commits so throughput can suffer.

   In this case, you need to make two lists:
   - One containing all the Tx's that have a commit record in the log. Redo them.
   - The other contains the Tx's that were active when the failure happened, but didn't commit, restart them.

Example: Consider the four TX's

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read_item(A)</td>
<td>Read_item(B)</td>
<td>Read_item(A)</td>
<td>Read_item(B)</td>
<td></td>
</tr>
<tr>
<td>Read_item(D)</td>
<td>Write_item(B)</td>
<td>Write_item(A)</td>
<td>Write_item(B)</td>
<td></td>
</tr>
<tr>
<td>Write_item(D)</td>
<td>Read_item(B)</td>
<td>Read_item(C)</td>
<td>Read_item(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write_item(D)</td>
<td>Write_item(C)</td>
<td>Write_item(A)</td>
<td></td>
</tr>
</tbody>
</table>

System log at the point of crash

Start(T1)
Write(T1,D,20)
Commit(T1)
Checkpoint
Start(T4)
Write(T4,B,15)
Write(T4,A,20)
Commit(T4)
Start(T2)
Write(T2,B,12)
Start(T3)
Write(T3,A,30)
Write(T2,D,25)
System crash

Recovery: T2 and T3 are ignored because they did not reach their commit points. T4 is redone because its commit point is after the last system checkpoint.

II. Using the immediate update protocol. Tx's may actually interfere with each other here.

1) Again make two lists:
First containing all Tx's committed since the last checkpoint.
Second containing all other active Tx's.
If a committed Tx has read from a value written by an active Tx, move the committed Tx to the set of active Txs.
2) Undo all the write_item operations of the active (uncommitted) Tx's, using the Undo procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3) Redo all the write_item operations of the remaining committed Tx's from the log, in the order in which they were written into the log. This can be done more efficiently by starting from the end of the log, and redoing only the last update of item X. Whenever an item is redone, it is added to the list of redone items and is not redone again.

E.g. the log:

/* T0 locks X */
/* T0 locks Y */
Read (T0, X)
Read (T0, Y)
Write (T0, X)
/* T0 unlocks X */
/* T1 locks X */
Read (T1, X)
Write (T1, X)
Commit (T1)
Write (T0, Y)
ERROR OCCURS HERE

Note that the technique called "shadow paging" requires no adaptation for a multi-user system, since the dirty page is kept locked until the Tx commits.