CS 242: Operating Systems
Transactions
Prasun Dewan

1 Atomic access to multiple objects

In our discussion of mutual exclusion so far, we have considered atomic access to a single object. What if we wish to atomically access multiple objects. For instance, what if we wish to atomically remove an item from the input buffer and put it in the echo buffer? The high-level constructs such as monitors and path expressions do not give users of shared objects flexibility in defining the unit of atomicity. They expect the programmers of shared objects to define the unit of atomicity. But the programmers do not know which other objects may be accessed concurrently (by object users) with the objects they are programming. Low-level mechanisms such as semaphores are not bound to a particular object, and thus can be used to provide multi-object atomicity. But these are hard to use and special care must be taken to ensure deadlocks are prevented.

Thus, what is needed is a higher-level abstraction for ensuring atomic access of multiples shared objects. The transaction is such an abstraction.

Trsnsaction models have been developed in the context of database management systems, operating systems, CAD tools, collaborative software engineering, and collaboration systems. We will focus here on the operating system models and the classical database models on which they are based.

Trsnsactions do not address both aspects of process coordination - they addresses mutual exclusion but not synchronization. However, in the context of transactions, ensuring mutual exclusion is called synchronization, which should not be confused with the producer-consumer synchronization we have seen so far.

2 Transaction Models

A transaction defines a set of "indivisible" steps, that is, commands with the Atomicity, Consistency, Isolation, and Durability (ACID) properties:
Atomicity: Either all or none of the steps of the transaction occur so that the invariants of the shared objects are maintained. A transaction is typically aborted by the system in response to failures but it may be aborted also by a user to "undo" the actions. In either case, the user is informed about the success or failure of the transaction.
Consistency: A transaction takes a shared object from one legal state to another, that is, maintains the invariant of the shared object.
Isolation: Events within a transaction may be hidden from other concurrently executing transactions. Techniques for achieving isolation are called synchronization schemes. They determine how these transactions are scheduled, that is, what the relationships are between the times the different steps of these transactions. Isolation is required to ensure that concurrent transactions do not cause an illegal state in the shared object and to prevent cascaded rollbacks when a transaction aborts.
Durability: Once the system tells the user that a transaction has completed successfully, it ensures that values written by the database system persist until they are explicitly overwritten by other transactions.

It is the isolation property that relates the transaction concept with concurrency control. Therefore, we will focus mainly on synchronization schemes, which we will also call concurrency schemes.

3 Serializability

Serializability is the classical concurrency scheme. It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations. A schedule is called “correct” if we can find a serial schedule that is “equivalent” to it. Given a set of transactions T1...Tn, two schedules S1 and S2 of these transactions are equivalent if the following conditions are satisfied:

Read-Write Synchronization: If a transaction reads a value written by another transaction in one schedule, then it also does so in the other schedule.

Write-Write Synchronization: If a transaction overwrites the value of another transaction in one schedule, it also does so in the other schedule.

These two properties ensure that there can be no difference in the effects of the two schedules. As an example, consider the schedule in Figure 1. It is equivalent to a schedule in which T2 is executed after T1.

T1	T2
read (p1)	
write(p1)	
	read (p1)
	write (p1)
read(p2)	
write(p2)	
	read(p2)
	write(p2)

Figure 1

The above definition of serializability states implies that if we draw a graph in which transactions are nodes and a directed edge is made from T1 to T2 if T1 reads/writes an item written by T2, or if T1 writes an item read by T2, then in a serializable schedule such a graph should not have cycles.

There are several approaches to enforcing serializability.

4 Locking

We could try and use locking to ensure serializability as shown in Figure 2.

Before accessing a resource, a process locks it, and after accessing it, the process unlocks it. A transaction blocks another one only if the latter concurrently accesses the same resource.

This approach works in the example shown in Figure 2 because both transactions access the shared objects in the same order. When this is not the case, it does not work. Consider the scenario of Figure 3.

This execution is allowed by the naive locking approach, but is not serializable.

Thus, a more complicated approach, described below, is necessary.

5 Two-Phase Locking

This approach, developed by Jim Gray, locks data and assumes that a transaction is divided into a growing phase, in which locks are only acquired, and a shrinking phase, in which locks are only released. The use of 2-phase locks is illustrated in Figure 4.

T1	T2
lock(p1)	
read(p1)	
	lock(p1)
write(p1)	
unlock (p1)	
lock(p2)	
	read (p1)
	write(p1)
	unlock(p1)
	lock(p2)
read(p2)	
write(p2)	
unlock (p2)	
	read(p2)
	write(p2)
	unlock(p2)

Figure 2

T1	T2
lock(p1)	
	lock(p2)
read(p1)	
	read (p2)
write(p1)	
	write(p2)
unlock (p1)	
	unlock(p2)
lock(p2)	
	lock(p1)
read(p2)	
	read(p1)
write(p2)	
	write(p2)
unlock (p2)	
	unlock(p1)

Figure 3

T1	T2
lock(p1)	
read(p1)	
	lock(p1)
write(p1)	
lock(p2)	
unlock (p1)	
	read (p1)
	write(p1)
	lock(p2)
	unlock(p1)
read(p2)	
write(p2)	
unlock (p2)	
	read(p2)
	write(p2)
	unlock(p2)

Figure 4

A transaction that tries to lock data that has been locked is forced to wait and may deadlock, as shown in Figure 5.

The above example illusrtates why 2PL ensures serializability. If a schedule is not serializable, transactions will deadlock if they try to execute a non serializable schedule. Recall that serializability required the transaction graph we created above to not have cycles. 2PL ensures that such cycles in the graph will lead to deadlocks rather than successful execution of the transactions. Let us try to prove this by contradiction. Suppose the transaction graph has cycles but there is no deadlock. If the graph has cycles then it means that we can find a pair of transactions. T1 and T2, such that T1 accesed an item, o1, after T2 accessed it, and T2 accessed an item, o2, before T accessed it. In 2PL we know that this means both transactions had locks to both items at some time. This means that one transaction had both locks before the other one had any lock. Let us say T1 had these locks before T2. As each lock serializes accesses to the item, this means T1 accessed both items before T2, which contradicts our claim that the transaction graph has cycles. Thus, cycles in the transaction graph result in cycles in cyclic locking.

6 Incremental Sharing

In Figure 4 and 5 above, transaction T2 was able to see incremental writes of T1. This approach increases the concurrency in the system but consider what happens if for some reason transaction T1 aborts. In this situation, the atomicity requirement implies that transactions such as T2 that have seen results of T1, transactions that have seen writes of these transactions, and so on must also be aborted. This problem is referred to as the problem of cascaded rollbacks, and locking schemes sometimes avoid this problem by unlocking data only at the end of the transaction.

T1	T2
lock(p1)	
	lock(p2)
read(p1)	
	read (p2)
write(p1)	
	write(p2)
lock(p2)	
	lock(p1)
unlock (p1)	
	unlock(p2)
read(p2)	
	read(p1)
write(p2)	
	write(p2)
unlock (p2)	
	unlock(p1)

Figure 5

7 Multiversion Timestamp Ordering

As we have seen above, a problem with 2PL is that it can lead to deadlocks. Reed's multiversion timestamp ordering scheme solves this problem by ordering transactions and aborting transactions that access data out of order. It also increases the concurrency in the system by never making an operation block (though it does abort transactions.)

2PL works because the lock status of items indicates the order in which transactions are currently accessing items, and locks are held long enough so that we can detect if serializablity is violated. Thus, if a transaction T2 tries to access an item T1 locked by another, then we know that T2 accesses the item after T1. As we saw earlier cyclic references in the transaction graph result in cylic locking.

Reed's scheme uses a more direct approach to determine the order in which items are accessed - timestamps. The basic idea is to assign transactions timestamps when they are started (which are used to order these transactions), keep with each data item information about the history of transactions that have accessed it so far and the kind of accesses they have made, and if two transactions access data items in an order that is inconsistent with their time stamps, then abort one of them.

Consider again the schedule of Figure 6.

Assume T1 entered the system before T2. Then the system will try to serialize them in this order. When transaction T1 tries to read p2 written earlier by T2, it aborts one of them as they are out of the serialization order being enforced.

A possible implementation of this approach is to require each data item to remember the timestamps of the last transactions that wrote and read the item. If a transaction with an earlier timestamp than the write timestamp tries to read or write to this item, then it is aborted. Similarly, if a transaction with an earlier timestamp than the read timestamp tries to write to this item, then it is aborted. This implementation would allow the schedule of Figure 1 but not Figure 6.

Unfortunately, this scheme is too conservative in comparison to Reed's scheme. The problem with it is

T1	T2
read(p1)	
	read (p2)
write(p1)	
	write(p2)
read(p2)	
	read(p1)
write(p2)	
	write(p2)

Figure 6

that it does not recognize that we can concurrently have transactions working with different versions of a data item that can be serialized.

Therefore, Reed’s algorithm is more complicated. As with the simpler scheme given above, it assigns transactions timestamps when they are started, which are used to order these transactions. Moreover, it associates each data item with timestamped versions and associates each version with readtimestamps. A readtimestamp is associated with a data item whenever a transaction reads the data item and is the same as the timestamp of the reading transaction. A timestamped version is created whenever a transaction writes a new value to the data item and has the timestamp of the writing transaction. The following steps occur when a transaction accesses the database:
If the operation is a read, then it is allowed, and the version read is the one with the largest timestamp less than the timestamp of the reading transaction. The timestamp of the reading transaction is added to the item.
If the operation is a write, then a new version of the data item is created with the timestamp of the writing transaction as long as no transaction with a more recent timestamp has read a version of the item with an older timestamp than that of the writing transaction. If this check fails, the writing transaction is aborted and restarted.

This approach will support the schedule shown in Figure 7 but not Figure 6. None of the schemes we have seen so far support the schedule of Figure 7. By making sure that T2 does not read the latest version of p, the scheme supports serializability. Perhaps even more interesting, it will allow the schedule of Figure

T1	T2	T3	T4
write(p)			
		write(p)	
	read (p)		
			read(p)

Figure 7

8. Here T2 writes the item after T3 writes it and T4 reads it, yet that is not a problem as no transaction after T2 has read a version with a time stamp earlier than T2. A scheme that kept a single time stamp with the object would not be able to allow this schedule.

Of course one problem with this scheme is that it tries only one serial order. Thus serializable schedule of Figure 9 is not allowed. The real problem with this scheme is the cost of keeping multiple versions and

T1	T2	T3	T4
write(p)			
		write(p)	
			read(p)
	write (p)		

Figure 8

associated time stamps. It is not clear when a version should be garbage collected.

T1	T2
	write (p)
read(p)	

Figure 9

8 Optimistic Concurrency Control

The previous schemes do incremental synchronization checks on each read/write– by using explicit locks or timestamps. There are several disadvantages of incremental checks:

Incremental checks can be expensive, specially when they involve accessing slow secondary memory.

They are an unnecessary overhead when there are no conflicts (consider readonly transactions).

They can reduce concurrency unnecessarily (because locks are kept longer than necessary to avoid cascaded aborts) or lead to cascaded aborts (consider Figure 1).

Optimistic concurrency control divides a transaction into a read phase, a validation phase, and a writing phase. During (a) the read phase, a transaction reads database items, and performs writes on local buffers, with no checking taking place, (b) validation phase, the system does synchronization checking, and (c) the write phase, the local writes are made global. It assigns each transaction a unique timestamp at the end of its read phase. A transaction TI is validated if one of the following conditions can be established for all transactions TJ with later timestamps:

Transaction TI completes its write phase before transaction TJ begins its read phase.

Transaction TJ does not read any of the items written by TI and transaction TI finishes its write phase before transaction TJ begins its write phase.

Transaction TJ does not read or write any items written by TI.

Transactions are aborted when validation cannot be done. This approach works well when there are no conflicts (hence the term optimistic) but wastes work when there are conflicts. Aborting of transactions is a severe problem when the transactions are long and interactive, when manual/automatic merging is a better alternative.

Optimistic transactions also have the disadvantage that they do not allow increment sharing of transaction actions, thereby limiting concurrency. For example, they cannot support the schedule of Figure 1 as the read phase of T2 begins before the end of the write phase of T1. On the other hand, they do not lead to cascaded aborts.

9 Variable Granularity Locking

In the design of a locking-based concurrency control mechanism for a hierarchical structure, the granularity of locking must be determined. One simple, popular approach is to support coarse-grained concurrency control which allows only one user to execute commands. In comparison to schemes that choose finer locking granularity, it is space efficient in that it stores only one lock per application, and time efficient in that it requires only one lock to be checked on each access. However, it limits the concurrency in the application when different components of an object can be manipulated independently. It is sometimes useful to support a compromise between fine-grain and coarse-grain locking by offering *variable-grained locking*. A simple method for supporting variable-grain locking is to allow transactions to lock both leaf and non-leaf nodes. A lock on a non-leaf node applies to all the leaf-level items in the subtree rooted by the node. We can define two types or modes of locks: shared locks and exclusive locks, which do not allow other transactions from writing and accessing, respectively, the locked data structure.

(In the discussion above, we assume that users operate only on leaf items. However, the solutions given above work for cases when the users can operate on non-leaf items.)

The variable-grained locking approach, as described above, has two related problems. First, it is inefficient in that when a transaction tries to lock a node, the system must check the lock status of all nodes in the subtree rooted by the node and the path from the node to the root. Second, it does not allow transactions to use exceptions in lock specifications such as lock all nodes in this tree with shared locks except this one in the exclusive mode. Transactions are forced to either take a conservative approach and lock the entire tree using a stronger lock than necessary or use a large number of locks.

Gray et al describe a scheme that addresses these problems. In addition to the basic or explicit locks, shared and exclusive, it associates non-leaf nodes with *intention locks*. An intention lock on a node served two purposes: First, it summarizes the locked status of its descendents, thereby reducing the need to check their individual locked status when the node is to be locked. Second, it allows exception-based specification. While a regular lock indicates that all descendent nodes are locked, an intention lock indicates that there exists a descendent node that is locked. Thus these two kinds of locks essentially assert the universal and existential operators respectively.

Now a transaction is required to put intention locks on all ancestors of a node before it puts a basic or explicit (shared or exclusive) lock on the node. Three kinds of intention locks are defined:

Intention Shared (IS) - it is put on a node if some descendent of the node is to be locked in the shared mode.

Intention Exclusive (IX) - it is put on a node if some descendent of the node is to be locked in the exclusive mode.

Shared Intention Exclusive (SIX) - it is put one a node if all children of a node are to be locked in the shared mode except for some children that are to be explicitly locked in the exclusive mode.

Thus, before putting a shared (exclusive) lock on a node, a transaction must put an IS or SIX (IX or SIX) lock on all parents of the node. Conversely, it must release the locks in a leaf to root order. When releasing a lock on a node it must ensure that the node lock status is restored, that is, the status reverts to the one it was before the transaction locked it. This can be done by keeping reference counts, checkpointing, or by keeping not one value but a list of values for a lock of a particular type.

The following compatibility matrix determines if a transaction can lock a node that is already locked by another transaction:

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

Compatibility Matrix

10 Nested Transactions

Imagine a software engineering project trying to fix a bug. One might want software engineers to fix two bugs in parallel but not create a new release until both bugs have been fixed and a check has been made that the two bug fixes do not interfere with each other. Regular transactions cannot support this scenario as units of parallel execution are also units of consistency. However, in our example, a fix to an individual bug does not necessarily leave the project in a consistent state as this fix may not work with the fix to the other bug.

The problem we have is that so far, our units of concurrency (transactions) have been required to also be units of atomicity, consistency, isolation, and durability. These properties are not orthogonal and several concurrency schemes support concurrency units (also called transactions) with only some of these properties

Nested transactions is one such concurrency scheme. It supports top-level transactions with all of the ACID properties. In addition, to support concurrent execution of independent actions (such as modification to two different procedures of a program) within these transactions, it allows a top-level transaction to root a tree of nested transactions. The following is an example of a top-level transaction with nested subtransactions T11 and T12.

```
T1:FixBug
T11: Fix Bug 1
T12: Fix Bug 2
Link Changed files with rest of the system
Run Tests
Fix Documentation
Mail to consumers
```

The transaction also contains basic commands such as Link and Send Mail. These commands are executed serially with respect to each other, much as statements in compound statement are exeuted serially with respect to each other. Should they also be executed serially with respect to the nested subtransactions or do they form a separate thread? In the above example, should the Link command wait for the two subtransactions to complete or be executed in parallel with them. Each of the alternatives has advantages. Allowing them to execute in parallel allows more concurrency while executing them serially allows top-level commands to be synchronized with nested transactions, which is what we want in this example.

Like top-level transactions, nested transactions have the following properties:
A transaction is serializable with respect to its siblings, that is, accesses to shared resources by sibling transactions have to obey the read-write and write-write synchronization rules.
A transaction is a unit of recovery, that is, it can be aborted independently of its siblings (modula the problem

of cascaded aborts).
A transaction is a unit of atomicity, that is, either all or none of the effects of its actions occur.

In addition, they have the following properties which stem from the fact that unlike top-level transactions they have parents:
A nested transaction's actions are not considered to conflict with its parent's actions. Thus, it can lock a resource locked by its parent as long as none of its siblings have locked it (in an incompatible mode).
A nested transaction can lock a datum in some mode only if its parent has locked the datum in the same or stronger mode.
A parent transaction's actions are considered to conflict with its child's actions but not vice versa. Thus, it cannot access a resource if a child's lock prohibits the access. Thus, the child's lock wins. (This is relevant only if the commands of the parent transaction can execute in parallel with the commands of the child transaction.)
An abort by a child transaction does not automatically abort the parent transaction. The parent is free to try alternative nested transactions. Thus, if a subtransaction accessing a particular server fails, another subtransaction accessing another server can be tried by the parent transaction.
A commit by a child transaction releases the locks held by it to its parent and makes its actions be part of the action set of its parent transaction. Thus, when the parent commits, it commits not only those actions it performed directly but also those performed by its descendents.

Notice, a nested transaction is not a unit of consistency or durability since it does not on its own leave the database in a consistent state.

11 Transactions on Objects

So far, we have considered concurrency schemes for synchronizing accesses to shared databases - that is, shared objects supporting only read/write operations. Object transactions are concurrency schemes for synchronizing accessed to shared objects on which arbitrary operations can be supported.

How should concurrency schemes be adapted when accesses are made to shared objects? The answer depends on how much of the application semantics is available to the concurrency scheme. To understand the nature of such a concurrency scheme, let us introduce the notion of a dependency relation, D (X,Y), which is a relation formed between transactions based on the order in which they execute certain kinds of operations, defined by X and Y, on a shared object. A transaction Tj depends on transaction Ti with respect to a dependency relation D(X,Y)

$$\text{Ti} < \text{D(X,Y) Tj}$$

if Ti performs an X operation before Tj performs a Y operation.

Consider first the situation when the concurrency scheme has no semantic information, that is, has no information regarding the effect of an operation on an object. In this situation, we must assume that an operation can perform arbitrary actions. Let us define one operation kind, any, and the dependency relation:

$$D = D(\text{any, any})$$

Then, a transaction schedule is serializable if this dependency relation does not introduce any cycles, that is, if transaction Ti performs an operation on an object before transaction Tj, then Tj does not perform an operation on the object before transaction Ti.

Also, assume that we wish to prevent cascaded aborts. Then we need to ensure that a transaction performs an operation on a shared object only if all transactions that have accessed the object have committed.

Now assume that we know which operations are reads and which are writes. We can define four dependency relations:

```
D1 = D(R, R)
D2 = D(R, W)
D3 = D(W, R)
D4 = D(W, W)
```

The dependency D1 is insignificant in that it cannot be observed. So to guarantee serializability of a schedule, we just have to ensure that it does not create cycles in the dependency relation D2 U D3 U D4, which is more liberal than D = D1 U D2 U D3 U D4. Thus, this scheme will allow the schedule of Figure 6 but not Figure 7.

T1	T2
R(O1)	
	R(O1)
W(O1)	

Figure 6

To ensure that there are no cascaded aborts, we have to ensure only that no D3 dependencies exist among overlapping transactions, which is again more liberal than ensuring no dependencies exist.

Finally, assume that we know type-specific semantics of the operations. In this case, it is possible that we can have an even more liberal scheme for ensuring serializability and no cascading aborts. Consider the example of a queue supporting the QEnter (q, e) (abbreviated as E(e)), and QDelete (q) (denoted below as D(e)), where e is the ID of the element inserted/deleted. Each element is assumed to have a unique ID assigned when it is entered into the queue. We can define the following dependency relations:

```
D1 = D (E(e), E(e’))
D2 = D (E(e), D(e’))
D3 = D (E(e), D(e))
D4 = D (D(e), E(e’))
D5 = D (D(e), D(e’))
```

In this case, the dependencies D2 and D4 are insignificant. Thus, assuming that queue, Q, has elements A and B, schedule of Figure 8 is serializable: If we were to model QEnter as a write and QDelete as a read followed by a write, then this schedule would not have been allowed. Since T1 $<$D2 T2 dependency is insignificant, this schedule is indeed serializable. Similarly, to prevent cascaded aborts, we have to ensure no D3 and D5 dependencies occur among interleaving transactions. The dependency D1 is insignificant since it is similar to the W,W dependency.

A locking-based scheme for supporting this scheme must support locks of the form LockClass(data) where LockClass corresponds to operation type and data corresponds to explicit or implicit operand of the operation. The compatibility matrix of Figure 9 describes a locking scheme for Queues: Here we assume, that QEnter must ask for a lock on the element to be added, which is held till the end of transaction to avoid

T1	T2
R(O1)	
	W(O1)
W(O1)	

Figure 7

T1	T2
E(C)	
	D(A)
E(D)	

Fugure 8

	E(e)	D(e)
E(e)	NA	NA
E(e’)	N	Y
D(e)	N	NA
D(e’)	Y	N

Fugure 9

cascaded aborts. Similarly, QDelete must ask for a lock on the head of the queue, which is also held until end of transaction. This scheme allows a transaction entering items to interleave its steps with one removing items.

We may further refine our transaction model by defining a weaker notion of consistency than serializability. This constraint is too strong in some situations. Let us return to our example of queues. We may want to use the queue as simply a buffer between producer and consumer processes. FIFO queues used as buffers ensure that items are removed in the order in which they are entered. We may be satisfied with weakly-FIFO queues– queues that allow items to be removed out of order but ensure there is no starvation - that is, an inserted item will eventually be removed by moving to the front of the queue and when it does so, it will be eventually removed. This may be useful, for example, when the buffer items are independent jobs to be executed by the consumers that are created by the producers. For such data structures, we can ignore the D1 and D5 conflicts, that is, we can allow transactions to overlap their entries and removals in a non-serial way. Thus, the only relationship we serialize is the D3 relationship to avoid cascaded aborts and to order transactions synchronizing on queue items. The locking scheme changes to Figure 10: This

	E(e)	D(e)
E(e)	NA	NA
E(e’)	Y	Y
D(e)	N	NA
D(e’)	Y	Y

Fugure 10

scheme allows transactions entering and deleting items to interleave their steps but is not serializable. To prevent cascaded aborts and order transactions synchonizing on a queue item, a transaction is not allowed to dequeue an element enqueued by an uncommitted transaction. A modified queue remove is supported to increase the concurrency in the system: If the entry at the front of the queue has been entered by an uncommitted transaction, then it is not removed to avoid cascaded aborts. Entries behind it are searched in the order in which they were entered for a committed one, and that is removed. If no committed entry is found, then the queue is searched for an entry made by the removing transaction, and that entry is removed.

If no such entry is also found, then the remove operation blocks.

This scheme allows:

entries made by a transaction to be interleaved in the queue,

dequeue order of items entered by the same transaction to be different from enqueue order.

The order depends on the commitment and abort order among (a) the enqueues, (b) dequeus, (c) dequeues and enqueues. As in a traditional queue, it also depends on the order in which the items were dequeued.

Let us take some examples:

Assume a queue is empty, and T1 adds item a and then T2 adds item b and then T1 adds item c. The queue is now: [a[1], b[2], c[1]], where i[j] says item i entered by Tj has not been committed. Now if they both commit, the queue is: (a, b, c). Thus, serializability of the queue is not maintained.

Now let us say transaction T3 removes the item at the front of the queue, and then T4 removes the next two items. The queue is now: [a(3), b(4), c(4)]. (i(j) says item i has been removed by uncommitted transaction Tj.) If now T3 aborts and T4 commits, the queue becomes: a. Thus, c is removed from the queue before a, even though it was inserted after it by the same transaction!

Consider another situation: [a[5], b[6]]. If T6 commits, the queue is: [a[5], b]. Now if T7 dequeues and commits, and then T5 commits, the queue is: [a]. Thus, as before, b is removed before a though it was added after it. The two items are removed in order in which their enquing transactions committed.

If in above, T6 commits after T5, but both commit before T7 dequeues, the queue is: [b]. Thus, items are not always removed in the enqueue commitment order.

Though items are removed out of order, no item remains in the Q forerever as long as:

the transaction that enters an item takes a finite amount of time to finish successfully (without abort).

a transaction that tries to remove an item takes a finite amont of time to finish successfully (without abort).

only a finite number of transactions that try and remove a queue item abort.