Lecture 25 — Deadlock, Lock Granularity

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 8, 2023

You should recall from learning about concurrency what deadlock is about, and therefore a brief review only is in order.

You may need to look through previous material (ECE 252) to get caught up.

Deadlock

We have already introduced the subject of deadlock and gave an informal definition as all transactions being "stuck" (unable to proceed).

A more formal definition: "the *permanent* blocking of a set of transactions that either compete for system resources or communicate with each other".

It may be possible for all transactions to be stuck temporarily, because one is waiting for some event (e.g., a read from disk).

Conditions for Deadlock

Remember, there are four condition necessary for a deadlock to take place:

1. **Mutual Exclusion**
2. **Hold-and-Wait**
3. **No Preemption**
4. **Circular-Wait**

Don’t Handle It

We could choose not to handle deadlock.

This option is certainly convenient for database system designers!

It is tempting and easy to just define a problem as being nothing we need to deal with, that’s unrealistic in reality.

If two transactions get deadlocked, the user may simply see no progress and just feel bad about it.

Deadlock Prevention

The first three conditions for deadlock (mutual exclusion, hold and wait, and no preemption) are all necessary for deadlock to be possible.

Goal: prevent circular wait from occurring.

We just cleverly avoid it taking place based on how we allow locks to occur.

Mutual Exclusion

This pillar cannot, generally speaking, be disallowed.

The purpose of mutual exclusion is to prevent errors (eg, inconsistent state)

Getting rid of mutual exclusion to rule out the possibility of deadlock is a cure that is worse than the disease.

It is therefore not acceptable as a solution.

Hold and Wait

To prevent the hold-and-wait condition, we must guarantee that when a transaction requests a resource, it does not have any other resource.

Previous idea: the program must request all resources at the beginning of execution.

Hold and Wait

So if the transaction is going to need resources R_1, R_2, and R_3 at some point?

All three must be requested at the beginning & held throughout the transaction.

No further resources may be requested at any time during execution.

No Preemption

If we violate this condition, it means that we do have preemption: forcible removal of resources from a transaction.

In the database context, if we have to preempt a transaction, it means the transaction in question is rolled back and then restarted.

To work out an ordering, transactions are assigned a timestamp.

If a transaction is rolled back, its timestamp remains the same.

There are two approaches for what happens if we need to do pre-emption:

The first is called wait-die.

If a transaction T_i requests an item held by T_j, then the timestamps of these two transactions are compared.

T_i will be allowed to wait if it is older (i.e. its timestamp is a smaller number) than T_j, otherwise it "dies" (is rolled back).

Or, Preemption

The second is wound-wait.

If a transaction T_i requests an item held by T_j, then the timestamps of these two transactions are compared.

If T_i is the younger transaction (its timestamp is larger), then T_i can wait.

Otherwise, T_j is “wounded” by T_i and T_j is rolled back.

Roll Back

So, in case of a conflict, one of the two transactions is going to be rolled back.

And it is always the younger one.

In either approach there may be unnecessary rollbacks...

Timeout

Instead we could also have locks that have timeouts.

When a lock is requested there is a max time the transaction is willing to wait.

If the limit is reached, the transaction rolls back automatically & begins again.

The difficulty lies in choosing the length of time that the transactions will wait.

Circular Wait

We have already discussed the idea of using two phase locking to put an ordering on the locks and we need not repeat that yet again.

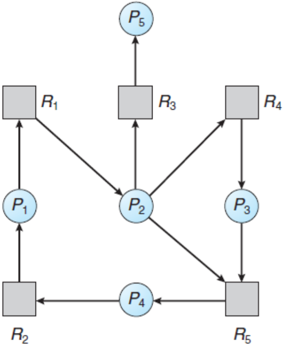
Deadlock Detection

A quick review of the graph algorithm follows, then.

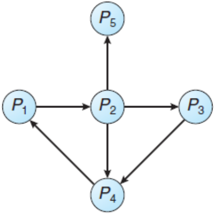
If resources have only a single instance, we may reduce the graph to a simplified version called the wait-for graph.

This removes the resource boxes from the diagram and indicates that a process P_i is waiting for process P_j rather than for a resource R_k that is held by P_j.

An edge $P_i \rightarrow P_j$ exists in the wait-for graph if and only if the resource allocation graph has a request $P_i \rightarrow R_k$ and an assignment edge $R_k \rightarrow P_j$.

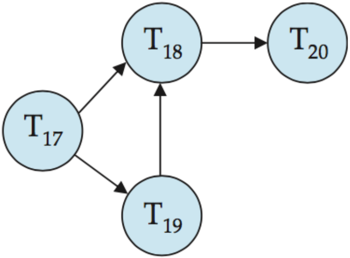


(a)



(b)

Because locks are unique then there is no need to use the resource allocation graph and we can always use the simplified wait-for graph for transactions:



Deadlock Detection is Expensive

Cycle detection algorithms tend to have runtime characteristics of $\Theta(n^2)$ where n is the number of nodes in the graph.

How often should the deadlock detection algorithm be run?

Deadlock Recovery

On the subject of what transaction to select as a victim we already identified the timestamp as one possible way to decide.

Other ideas?

1. How long the transaction has been executing.
2. How long is remaining in execution
3. What resources the transaction has used.
4. Future resource requests, if known.
5. How many times, if any, the transaction has been selected as a victim.

Push the Ball Uphill

We chose a victim, now simply roll that transaction back!

That would be a total rollback: undo all the steps of the transaction and start from the beginning once again.

There exists, however, the ability to do a partial rollback.

Partial Rollback

The goal of the partial rollback is to rollback the transaction back only as much as is necessary to break the deadlock.

This is accomplished by maintaining the sequence of lock requests and grants.

By finding the lock(s) that are implicated in the deadlock, we can roll back a transaction to where it obtained the first of such locks.

Lock Granularity

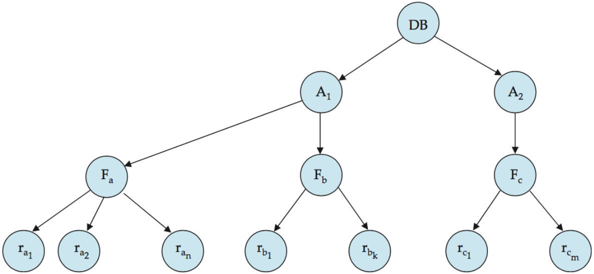
Locks' extents constitute their granularity: that is, how much data is being protected by such a lock.

Coarse-grained locking is easier to implement, but it can significantly reduce opportunities for parallelism.

Fine-grained locking requires more careful design, increases locking overhead and is more prone to bugs (deadlock etc).

Granularity Hierarchy

In the database we could define multiple levels of granularity.



Intention Lock Modes

A more efficient route is intention lock modes, which are put on the ancestors of a node before the node is locked.

These intention locks can be checked while traversing the tree. The following modes are possible:

- **Shared** (S)
- **Exclusive** (X)
- **Intention Shared** (IS)
- **Intention Exclusive** (IX)
- **Shared Intention Exclusive** (SIX)

Intention Lock Rules

Then a transaction must follow the six rules below:

1. All attempts to lock must observe the table (below) as to whether they are permitted to proceed (or wait).
2. The transaction must lock the root of the tree first (in any mode, but it must pick one).
3. A node *n* can be locked in **S** or **IS** mode only if the parent of *n* is locked in either **IX** or **IS** mode.

Intention Lock Rules

Then a transaction must follow the six rules below:

4. A node *n* can be locked in **X**, **SIX**, or **IX** mode only if the parent of *n* is locked in either **IX** or **SIX** mode.
5. A node *n* may only be locked if the transaction has not previously unlocked any nodes.
6. A node *n* may only be unlocked if none of the children of *n* are locked by this transaction.

Locks must be acquired in a top-down manner and then released in a bottom-up manner.

Lock Compatibility Matrix

Compatibility	**IS**	**IX**	**S**	**SIX**	**X**
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×