3 TWO PHASE LOCKING

3.1 AGGRESSIVE AND CONSERVATIVE SCHEDULERS

In this chapter we begin our study of practical schedulers by looking at two phase locking schedulers, the most popular type in commercial products. For most of the chapter, we focus on locking in centralized DBSs, using the model presented in Chapter 1. Later sections show how locking schedulers can be modified to handle a distributed system environment. The final section discusses specialized locking protocols for trees and dags.

Recall from Chapter 1 that when a scheduler receives an operation from a TM it has three options:

1. immediately schedule it (by sending it to the DM);
2. delay it (by inserting it into some queue); or
3. reject it (thereby causing the issuing transaction to abort).

Each type of scheduler usually favors one or two of these options. Based on which of these options the scheduler favors, we can make the fuzzy, yet conceptually useful, distinction between *aggressive* and *conservative* schedulers.

An aggressive scheduler tends to avoid delaying operations; it tries to schedule them immediately. But to the extent it does so, it foregoes the opportunity to reorder operations it receives later on. By giving up the opportunity to reorder operations, it may get stuck in a situation in which it has no hope of finishing the execution of all active transactions in a serializable fashion. At this point, it has to resort to rejecting operations of one or more transactions, thereby causing them to abort (option (3) above).

A conservative scheduler, on the other hand, tends to delay operations. This gives it more leeway to reorder operations it receives later on. This leeway makes it less likely to get stuck in a situation where it has to reject operations to produce an SR execution. An extreme case of a conservative scheduler is one that, at any given time, delays the operations of all but one transaction. When that transaction terminates, another one is selected to have its operations processed. Such a scheduler processes transactions serially. It never needs to reject an operation, but avoids such rejections by sometimes excessively delaying operations.

There is an obvious performance trade-off between aggressive and conservative schedulers. Aggressive schedulers avoid delaying operations and thereby risk rejecting them later. Conservative schedulers avoid rejecting operations by deliberately delaying them. Each approach works especially well for certain types of applications.

For example, in an application where transactions that are likely to execute concurrently rarely conflict, an aggressive scheduler might perform better than a conservative one. Since conflicts are rare, conflicts that require the rejection of an operation are even rarer. Thus, the aggressive scheduler would not reject operations very often. By contrast, a conservative scheduler would needlessly delay operations, anticipating conflicts that seldom materialize.

On the other hand, in an application where transactions that are likely to execute concurrently conflict, a conservative scheduler's cautiousness may pay off. An aggressive scheduler might output operations recklessly, frequently placing itself in the undesirable position where rejecting operations is the only alternative to producing incorrect executions.

The rate at which conflicting operations are submitted is not the only factor that affects concurrency control performance. For example, the load on computer resources other than the DBS is also important. Therefore, this discussion of trade-offs between aggressive and conservative approaches to scheduling should be taken with a grain of salt. The intent is to develop some intuition about the operation of schedulers, rather than to suggest precise rules for designing them. Unfortunately, giving such precise rules for tailoring a scheduler to the performance specifications of an application is beyond the state-of-the-art.

Almost all types of schedulers have an aggressive and a conservative version. Generally speaking, a conservative scheduler tries to anticipate the future behavior of transactions in order to prepare for operations that it has not yet received. The main information it needs to know is the set of data items that each transaction will read and write (called, respectively, the *readset* and *writeset* of the transaction). In this way, it can predict which of the operations that it is currently scheduling may conflict with operations that will arrive in the future. By contrast, an aggressive scheduler doesn't need this information, since it schedules operations as early as it can, relying on rejections to correct mistakes.

A very conservative version of any type of scheduler can usually be built if transactions *predeclare their readsets and writesets.* This means that the TM begins processing a transaction by giving the scheduler the transaction's readset and writeset. Predeclaration is more easily and efficiently done if transactions are analyzed by a preprocessor, such as a compiler, before being submitted to the system, rather than being interpreted on the fly.

An impediment to building very conservative schedulers is that different executions of a given program may result in transactions that access different sets of data items. This occurs if programs contain conditional statements. For example, the following program reads either x and y, or x and z, depending on the value of x that it reads.

```
Procedure Fuzzy-readset begin
    Start;
    $a$ := Read($x$);
    if ($a > 0$) then $b$ := Read($y$) else $b$ := Read($z$);
    Commit
end
```

In this case the transaction must predeclare the set of all data items it *might* read or write. This often causes the transaction to overstate its readset and writeset. For example, a transaction executing Fuzzy-readset would declare its readset to be $\{x, y, z\}$, even though on any single execution it will only access two of those three data items. The same problem may occur if transactions interact with the DBS using a high level (e.g., relational) query language. A high level query may *potentially* access large portions of the database, even though on any *single* execution it only accesses a small portion of the database. When transactions overstate readsets and writesets, the scheduler ends up being even more conservative than it has to be, since it will delay certain operations in anticipation of others that will never be issued.

3.2 BASIC TWO PHASE LOCKING

Locking is a mechanism commonly used to solve the problem of synchronizing access to shared data. The idea behind locking is intuitively simple. Each data item has a *lock* associated with it. Before a transaction T_1 may access a data item, the scheduler first examines the associated lock. If no transaction holds the lock, then the scheduler obtains the lock on behalf of T_1. If another transaction T_2 does hold the lock, then T_1 has to wait until T_2 gives up the lock. That is, the scheduler will not give T_1 the lock until T_2 releases it. The scheduler thereby ensures that only one transaction can hold the lock at a time, so only one transaction can access the data item at a time.

Locking can be used by a scheduler to ensure serializability. To present such a locking protocol, we need some notation.

Transactions access data items either for reading or for writing them. We therefore associate *two* types of locks with data items: read locks and write locks. We use $rl[x]$ to denote a read lock on data item x and $wl[x]$ to denote a write lock on x. We use $rl_i[x]$ (or $wl_i[x]$) to indicate that transaction T_i has obtained a read (or write) lock on x. As in Chapter 2, we use the letters o, p, and q to denote an arbitrary type of operation, that is, a Read (r) or Write (w). We use $ol_i[x]$ to denote a lock of type o by T_i on x.

Locks can be thought of as entries in a lock table. For example, $rl_i[x]$ corresponds to the entry $[x, r, T_i]$ in the table. For now, the detailed data structure of the table is unimportant. We'll discuss those details in Section 3.6.

Two locks $pl_i[x]$ and $ql_j[y]$ *conflict* if $x = y$, $i \neq j$, and operations p and q are of conflicting type. That is, two locks conflict if they are on the same data item, they are issued by different transactions, and one or both of them are write locks.[1] Thus, two locks on different data items do not conflict, nor do two locks that are on the same data item and are owned by the same transaction, even if they are of conflicting type.

We also use $rl_i[x]$ (or $wl_i[x]$) to denote the *operation* by which T_i *sets* or *obtains* a read (or write) lock on x. It will always be clear from the context whether $rl_i[x]$ and $wl_i[x]$ denote locks or operations that set locks.

We use $ru_i[x]$ (or $wu_i[x]$) to denote the operation by which T_i *releases* its read (or write) lock on x. In this case, we say T_i *unlocks* x (the u in ru and wu means unlock).

It is the job of a two phase locking (2PL) scheduler to manage the locks by controlling when transactions obtain and release their locks. In this section, we'll concentrate on the *Basic* version of 2PL. We'll look at specializations of 2PL in later sections.

Here are the rules according to which a Basic 2PL scheduler manages and uses its locks:

1. When it receives an operation $p_i[x]$ from the TM, the scheduler tests if $pl_i[x]$ conflicts with some $ql_j[x]$ that is already set. If so, it delays $p_i[x]$, forcing T_i to wait until it can set the lock it needs. If not, then the scheduler sets $pl_i[x]$, and then sends $p_i[x]$ to the DM.[2]
2. Once the scheduler has set a lock for T_i, say $pl_i[x]$, it may not release that lock *at least* until after the DM acknowledges that it has processed the lock's corresponding operation, $p_i[x]$.
3. Once the scheduler has released a lock for a transaction, it may not subsequently obtain *any* more locks for that transaction (on *any* data item).

[1]We will generalize the notion of lock conflict to operations other than Read and Write in Section 3.8.

[2]The scheduler must be implemented so that setting a lock is atomic relative to setting conflicting locks. This ensures that conflicting locks are never held simultaneously.

Rule (1) prevents two transactions from concurrently accessing a data item in conflicting modes. Thus, conflicting operations are scheduled in the same order in which the corresponding locks are obtained.

Rule (2) supplements rule (1) by ensuring that the DM processes operations on a data item in the order that the scheduler submits them. For example, suppose T_i obtains $rl_i[x]$, which it releases before the DM has confirmed that $r_i[x]$ has been processed. Then it is possible for T_j to obtain a conflicting lock on x, $wl_j[x]$, and send $w_j[x]$ to the DM. Although the scheduler has sent the DM $r_i[x]$ before $w_j[x]$, without rule (2) there is no guarantee that the DM will receive and process the operations in that order.

Rule (3), called the *two phase rule*, is the source of the name *two phase locking*. Each transaction may be divided into two phases: a *growing phase* during which it obtains locks, and a *shrinking phase* during which it releases locks. The intuition behind rule (3) is not obvious. Roughly, its function is to guarantee that *all* pairs of conflicting operations of two transactions are scheduled in the same order. Let's look at an example to see, intuitively, why this might be the case.

Consider two transactions T_1 and T_2:

$$T_1\colon r_1[x] \rightarrow w_1[y] \rightarrow c_1 \qquad T_2\colon w_2[x] \rightarrow w_2[y] \rightarrow c_2$$

and suppose they execute as follows:

$$H_1 = rl_1[x]\ r_1[x]\ ru_1[x]\ wl_2[x]\ w_2[x]\ wl_2[y]\ w_2[y]\ wu_2[x]\ wu_2[y]\ c_2\ wl_1[y]\ w_1[y]\ wu_1[y]\ c_1$$

Since $r_1[x] < w_2[x]$ and $w_2[y] < w_1[y]$, SG(H_1) consists of the cycle $T_1 \rightarrow T_2 \rightarrow T_1$. Thus, H_1 is not SR.

The problem in H_1 is that T_1 released a lock ($ru_1[x]$) and subsequently set a lock ($wl_1[y]$), in violation of the two phase rule. Between $ru_1[x]$ and $wl_1[y]$, another transaction T_2 wrote into both x and y, thereby appearing to follow T_1 with respect to x and precede it with respect to y. Had T_1 obeyed the two phase rule, this "window" between $ru_1[x]$ and $wl_1[y]$ would not have opened, and T_2 could not have executed as it did in H_1. For example, T_1 and T_2 might have executed as follows.

1. Initially, neither transaction owns any locks.
2. The scheduler receives $r_1[x]$ from the TM. Accordingly, it sets $rl_1[x]$ and submits $r_1[x]$ to the DM. Then the DM acknowledges the processing of $r_1[x]$.
3. The scheduler receives $w_2[x]$ from the TM. The scheduler can't set $wl_2[x]$, which conflicts with $rl_1[x]$, so it delays the execution of $w_2[x]$ by placing it on a queue.
4. The scheduler receives $w_1[y]$ from the TM. It sets $wl_1[y]$ and submits $w_1[y]$ to the DM. Then the DM acknowledges the processing of $w_1[y]$.
5. The scheduler receives c_1 from the TM, signalling that T_1 has terminated. The scheduler sends c_1 to the DM. After the DM acknowledges

processing c_1, the scheduler releases $rl_1[x]$ and $wl_1[y]$. This is safe with respect to rule (2), because $r_1[x]$ and $w_1[y]$ have already been processed, and with respect to rule (3), because T_1 won't request any more locks.

6. The scheduler sets $wl_2[x]$ so that $w_2[x]$, which had been delayed, can now be sent to the DM. Then the DM acknowledges $w_2[x]$.
7. The scheduler receives $w_2[y]$ from the TM. It sets $wl_2[y]$ and sends $w_2[y]$ to the DM. The DM then acknowledges processing $w_2[y]$.
8. T_2 terminates and the TM sends c_2 to the scheduler. The scheduler sends c_2 to the DM. After the DM acknowledges processing c_2, the scheduler releases $wl_2[x]$ and $wl_2[y]$.

This execution is represented by the following history.

$$H_2 = rl_1[x]\ r_1[x]\ wl_1[y]\ w_1[y]\ c_1\ ru_1[x]\ wu_1[y]\ wl_2[x]\ w_2[x]\ wl_2[y]\ w_2[y]\ c_2\ wu_2[x]\ wu_2[y].$$

H_2 is serial and therefore is SR.

An important and unfortunate property of 2PL schedulers is that they are subject to *deadlocks*. For example, suppose a 2PL scheduler is processing transactions T_1 and T_3

$$T_1\colon r_1[x] \rightarrow w_1[y] \rightarrow c_1 \qquad T_3\colon w_3[y] \rightarrow w_3[x] \rightarrow c_3$$

and consider the following sequence of events:

1. Initially, neither transaction holds any locks.
2. The scheduler receives $r_1[x]$ from the TM. It sets $rl_1[x]$ and submits $r_1[x]$ to the DM.
3. The scheduler receives $w_3[y]$ from the TM. It sets $wl_3[y]$ and submits $w_3[y]$ to the DM.
4. The scheduler receives $w_3[x]$ from the TM. The scheduler does not set $wl_3[x]$ because it conflicts with $rl_1[x]$ which is already set. Thus $w_3[x]$ is delayed.
5. The scheduler receives $w_1[y]$ from the TM. As in (4), $w_1[y]$ must be delayed.

Although the scheduler behaved exactly as prescribed by the rules of 2PL schedulers, neither T_1 nor T_3 can complete without violating one of these rules. If the scheduler sends $w_1[y]$ to the DM without setting $wl_1[y]$, it violates rule (1). Similarly for $w_3[x]$. Suppose the scheduler releases $wl_3[y]$, so it can set $wl_1[y]$ and thereby be allowed to send $w_1[y]$ to the DM. In this case, the scheduler will never be able to set $wl_3[x]$ (so it can process $w_3[x]$), or else it would violate rule (3). Similarly if it releases $rl_1[x]$. The scheduler has painted itself into a corner.

This is a classic deadlock situation. Before either of two processes can proceed, one must release a resource that the other needs to proceed.

Deadlock also arises when transactions try to strengthen read locks to write locks. Suppose a transaction T_i reads a data item x and subsequently tries to write it. T_i issues $r_i[x]$ to the scheduler, which sets $rl_i[x]$. When T_i issues $w_i[x]$ to the scheduler, the scheduler must upgrade $rl_i[x]$ to $wl_i[x]$. This upgrading of a lock is called a lock *conversion*. To obey 2PL, the scheduler must not release $rl_i[x]$. This is not a problem, because locks set by the same transaction do not conflict with each other. However, if two transactions concurrently try to convert their read locks on a data item into write locks, the result is deadlock.

For example, suppose T_4 and T_5 issue operations to a 2PL scheduler.

T_4: $r_4[x] \rightarrow w_4[x] \rightarrow c_4$ $\quad$ T_5: $r_5[x] \rightarrow w_5[x] \rightarrow c_5$

The scheduler might be confronted with the following sequence of events:

1. The scheduler receives $r_4[x]$, and therefore sets $rl_4[x]$ and sends $r_4[x]$ to the DM.
2. The scheduler receives $r_5[x]$, and therefore sets $rl_5[x]$ and sends $r_5[x]$ to the DM.
3. The scheduler receives $w_4[x]$. It must delay the operation, because $wl_4[x]$ conflicts with $rl_5[x]$.
4. The scheduler receives $w_5[x]$. It must delay the operation, because $wl_5[x]$ conflicts with $rl_4[x]$.

Since neither transaction can release the $rl[x]$ it owns, and since neither can proceed until it sets $wl[x]$, the transactions are deadlocked. This type of deadlock commonly occurs when a transaction scans a large number of data items looking for data items that contain certain values, and then updates those data items. It sets a read lock on each data item it scans, and converts a read lock into a write lock only when it decides to update a data item.

We will examine ways of dealing with deadlocks in Section 3.4.

3.3 *CORRECTNESS OF BASIC TWO PHASE LOCKING

To prove that a scheduler is correct, we have to prove that all histories representing executions that could be produced by it are SR. Our strategy for proving this has two steps. First, given the scheduler we characterize the properties that all of its histories must have. Second, we prove that any history with these properties must be SR. Typically this last part involves the Serializability Theorem. That is, we prove that for any history H with these properties, SG(H) is acyclic.

To prove the correctness of the 2PL scheduler, we must characterize the set of *2PL histories*, that is, those that represent possible executions of transactions that are synchronized by a 2PL scheduler. To characterize 2PL histories, we'll find it very helpful to include the Lock and Unlock operations. (They were not in our formal model of Chapter 2.) Examining the order in which Lock and Unlock operations are processed will help us establish the order in which Reads and Writes are executed. This, in turn, will enable us to prove that the SG of any history produced by 2PL is acyclic.

To characterize 2PL histories, let's list all of the orderings of operations that we know must hold. First, we know that a lock is obtained for each database operation before that operation executes. This follows from rule (1) of 2PL. That is, $ol_i[x] < o_i[x]$. From rule (2) of 2PL, we know that each operation is executed by the DM before its corresponding lock is released. In terms of histories, that means $o_i[x] < ou_i[x]$. In particular, if $o_i[x]$ belongs to a committed transaction (all of whose operations are therefore in the history), we have $ol_i[x] < o_i[x] < ou_i[x]$.

Proposition 3.1: Let H be a history produced by a 2PL scheduler. If $o_i[x]$ is in C(H), then $ol_i[x]$ and $ou_i[x]$ are in C(H), and $ol_i[x] < o_i[x] < ou_i[x]$. □

Now suppose we have two operations $p_i[x]$ and $q_j[x]$ that conflict. Thus, the locks that correspond to these operations also conflict. By rule (1) of 2PL, only one of these locks can be held at a time. Therefore, the scheduler must release the lock corresponding to one of the operations before it sets the lock for the other. In terms of histories, we must have $pu_i[x] < ql_j[x]$ or $qu_j[x] < pl_i[x]$.

Proposition 3.2: Let H be a history produced by a 2PL scheduler. If $p_i[x]$ and $q_j[x]$ $(i \neq j)$ are conflicting operations in C(H), then either $pu_i[x] < ql_j[x]$ or $qu_j[x] < pl_i[x]$. □

Finally, let's look at the two phase rule, which says that once a transaction releases a lock it cannot subsequently obtain any other locks. This is equivalent to saying that every lock operation of a transaction executes before every unlock operation of that transaction. In terms of histories, we can write this as $pl_i[x] < qu_i[y]$.

Proposition 3.3: Let H be a complete history produced by a 2PL scheduler. If $p_i[x]$ and $q_i[y]$ are in C(H), then $pl_i[x] < qu_i[y]$. □

Using these properties, we must now show that every 2PL history H has an acyclic SG. The argument has three steps. (Recall that SG(H) contains nodes only for the committed transactions in H.)

1. If $T_i \rightarrow T_j$ is in SG(H), then one of T_i's operations on some data item, say x, executed before and conflicted with one of T_j's operations. Therefore, T_i must have released its lock on x *before* T_j set its lock on x.

2. Suppose $T_i \rightarrow T_j \rightarrow T_k$ is a path in SG(H). From step (1), T_i released some lock before T_j set some lock, and similarly T_j released some lock before T_k set some lock. Moreover, by the two phase rule, T_j set all of its locks before it released any of them. Therefore, by transitivity, T_i released some lock before T_k set some lock. By induction, this argument extends to arbitrarily long paths in SG(H). That is, for any path $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$, T_1 released some lock before T_n set some lock.
3. Suppose SG(H) had a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$. Then by step (2), T_1 released a lock before T_1 set a lock. But then T_1 violated the two phase rule, which contradicts the fact that H is a 2PL history. Therefore, the cycle cannot exist. Since SG(H) has no cycles, the Serializability Theorem implies that H is SR.

Notice that in step (2), the lock that T_i released does not necessarily conflict with the one that T_k set, and in general they do not. T_i's lock conflicts with and precedes one that T_j set, and T_j released a lock (possibly a different one) that conflicts with and precedes the one that T_k set. For example, the history that leads to the path $T_i \rightarrow T_j \rightarrow T_k$ could be

$$r_i[x] \rightarrow w_j[x] \rightarrow w_j[y] \rightarrow r_k[y].$$

T_i's lock on x does not conflict with T_k's lock on y.

We formalize this three step argument in the following lemmas and theorem. The two lemmas formalize steps (1) and (2). The theorem formalizes step (3).

Lemma 3.4: Let H be a 2PL history, and suppose $T_i \rightarrow T_j$ is in SG(H). Then, for some data item x and some conflicting operations $p_i[x]$ and $q_j[x]$ in H, $pu_i[x] < ql_j[x]$.

Proof: Since $T_i \rightarrow T_j$, there must exist conflicting operations $p_i[x]$ and $q_j[x]$ such that $p_i[x] < q_j[x]$. By Proposition 3.1,

1. $pl_i[x] < p_i[x] < pu_i[x]$, and
2. $ql_j[x] < q_j[x] < qu_j[x]$.

By Proposition 3.2, either $pu_i[x] < ql_j[x]$ or $qu_j[x] < pl_i[x]$. In the latter case, by (1), (2) and transitivity, we would have $q_j[x] < p_i[x]$, which contradicts $p_i[x] < q_j[x]$. Thus, $pu_i[x] < ql_j[x]$, as desired. □

Lemma 3.5: Let H be a 2PL history, and let $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ be a path in SG(H), where $n > 1$. Then, for some data items x *and* y, and some operations $p_1[x]$ and $q_n[y]$ in H, $pu_1[x] < ql_n[y]$.

Proof: The proof is by induction on n. The basis step, for $n = 2$, follows immediately from Lemma 3.4.

For the induction step, suppose the lemma holds for $n = k$ for some $k \geq 2$. We will show that it holds for $n = k + 1$. By the induction hypothesis, the path $T_1 \rightarrow \cdots \rightarrow T_k$ implies that there exist data items x and z, and operations $p_1[x]$ and $o_k[z]$ in H, such that $pu_1[x] < ol_k[z]$. By $T_k \rightarrow T_{k+1}$ and Lemma 3.4, there exists data item y and conflicting operations $o'_k[y]$, and $q_{k+1}[y]$ in H, such that $o'u_k[y] < ql_{k+1}[y]$. By Proposition 3.3, $ol_k[z] < o'u_k[y]$. By the last three precedences and transitivity, $pu_1[x] < ql_{k+1}[y]$, as desired. □

Theorem 3.6: Every 2PL history H is serializable.

Proof: Suppose, by way of contradiction, that SG(H) contains a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 3.5, for some data items x and y, and some operations $p_1[x]$ and $q_1[y]$ in H, $pu_1[x] < ql_1[y]$. But this contradicts Proposition 3.3. Thus, SG(H) has no cycles and so, by the Serializability Theorem, H is SR. □

3.4 DEADLOCKS

The scheduler needs a strategy for detecting deadlocks, so that no transaction is blocked forever. One strategy is *timeout*. If the scheduler finds that a transaction has been waiting too long for a lock, then it simply guesses that there may be a deadlock involving this transaction and therefore aborts it. Since the scheduler is only guessing that a transaction may be involved in a deadlock, it may be making a mistake. It may abort a transaction that isn't really part of a deadlock but is just waiting for a lock owned by another transaction that is taking a long time to finish. There's no harm done by making such an incorrect guess, insofar as correctness is concerned. There *is* certainly a performance penalty to the transaction that was unfairly aborted, though as we'll see in Section 3.12, the overall effect may be to improve transaction throughput.

One can avoid too many of these types of mistakes by using a long timeout period. The longer the timeout period, the more chance that the scheduler is aborting transactions that are actually involved in deadlocks. However, a long timeout period has a liability, too. The scheduler doesn't notice that a transaction might be deadlocked until the timeout period has elapsed. So, should a transaction become involved in a deadlock, it will lose some time waiting for its deadlock to be noticed. The timeout period is therefore a parameter that needs to be tuned. It should be long enough so that most transactions that are aborted are actually deadlocked, but short enough that deadlocked transactions don't wait too long for their deadlocks to be noticed. This tuning activity is tricky but manageable, as evidenced by its use in several commercial products, such as Tandem.

Another approach to deadlocks is to detect them precisely. To do this, the scheduler maintains a directed graph called a *waits-for graph* (*WFG*). The nodes of WFG are labelled with transaction names. There is an edge $T_i \rightarrow T_j$, from node T_i to node T_j, iff transaction T_i is waiting for transaction T_j to release some lock.[3]

Suppose a WFG has a cycle: $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$. Each transaction is waiting for the next transaction in the cycle. So, T_1 is waiting for itself, as is every other transaction in the cycle. Since all of these transactions are blocked waiting for locks, none of the locks they are waiting for are ever going to be released. Thus, the transactions are deadlocked. Exploiting this observation, the scheduler can detect deadlocks by checking for cycles in WFG.

Of course, the scheduler has to maintain a representation of the WFG in order to check for cycles in it. The scheduler can easily do this by adding an edge $T_i \rightarrow T_j$ to the WFG whenever a lock request by T_i is blocked by a conflicting lock owned by T_j. It drops an edge $T_i \rightarrow T_j$ from the WFG whenever it releases the (last) lock owned by T_j that had formerly been blocking a lock request issued by T_i. For example, suppose the scheduler receives $r_i[x]$, but has to delay it because T_j already owns $wl_j[x]$. Then it adds an edge $T_i \rightarrow T_j$ to the WFG. After T_j releases $wl_j[x]$, the scheduler sets $rl_i[x]$, and therefore deletes the edge $T_i \rightarrow T_j$.

How often should the scheduler check for cycles in the WFG? It could check every time a new edge is added, looking for cycles that include this new edge. But this could be quite expensive. For example, if operations are frequently delayed, but deadlocks are relatively rare, then the scheduler is spending a lot of effort looking for deadlocks that are hardly ever there. In such cases, the scheduler should check for cycles less often. Instead of checking every time an edge is added, it waits until a few edges have been added, or until some timeout period has elapsed. There is no danger in checking less frequently, since the scheduler will never miss a deadlock. (Deadlocks don't go away by themselves!) Moreover, by checking less frequently, the scheduler incurs the cost of cycle detection less often. However, a deadlock may go undetected for a longer period this way. In addition, *all* cycles must be found, not just those involving the most recently added edge.

When the scheduler discovers a deadlock, it must break the deadlock by aborting a transaction. The Abort will in turn delete the transaction's node from the WFG. The transaction that it chooses to abort is called the *victim*. Among the transactions involved in a deadlock cycle in WFG, the scheduler should select a victim whose abortion costs the least. Factors that are commonly used to make this determination include:

[3]WFGs are related to SGs in the following sense. If $T_i \rightarrow T_j$ is in the WFG, and both T_i and T_j ultimately commit, then $T_j \rightarrow T_i$ will be in the SG. However, if T_i aborts, then $T_j \rightarrow T_i$ may never appear in the SG. That is, WFGs describe the current state of transactions, which includes waits-for situations involving operations that never execute (due to abortions). SGs only describe dependencies between committed transactions (which arise from operations that actually execute).

- The amount of effort that has already been invested in the transaction. This effort will be lost if the transaction is aborted.
- The cost of aborting the transaction. This cost generally depends on the number of updates the transaction has already performed.
- The amount of effort it will take to finish executing the transaction. The scheduler wants to avoid aborting a transaction that is almost finished. To do this, it must be able to predict the future behavior of active transactions, e.g., based on the transaction's type (Deposits are short, Audits are long).
- The number of cycles that contain the transaction. Since aborting a transaction breaks all cycles that contain it, it is best to abort transactions that are part of more than one cycle (if such transactions exist).

A transaction can repeatedly become involved in deadlocks. In each deadlock, the transaction is selected as the victim, aborts, and restarts its execution, only to become involved in a deadlock again. To avoid such *cyclic restarts*, the victim selection algorithm should also consider the number of times a transaction is aborted due to deadlock. If it has been aborted too many times, then it should not be a candidate for victim selection, unless all transactions involved in the deadlock have reached this state.

3.5 VARIATIONS OF TWO PHASE LOCKING

Conservative 2PL

It is possible to construct a 2PL scheduler that never aborts transactions. This technique is known as *Conservative 2PL* or *Static 2PL*. As we have seen, 2PL causes abortions because of deadlocks. Conservative 2PL avoids deadlocks by requiring each transaction to obtain *all* of its locks before *any* of its operations are submitted to the DM. This is done by having each transaction predeclare its readset and writeset. Specifically, each transaction T_i first tells the scheduler all the data items it will want to Read or Write, for example as part of its Start operation. The scheduler tries to set *all* of the locks needed by T_i. It can do this providing that none of these locks conflicts with a lock held by any other transaction. If the scheduler succeeds in setting all of T_i's locks, then it submits T_i's operations to the DM as soon as it receives them. After the DM acknowledges the processing of T_i's last database operation, the scheduler may release all of T_i's locks.

If, on the other hand, *any* of the locks requested in T_i's Start conflicts with locks presently held by other transactions, then the scheduler does not grant any of T_i's locks. Instead, it inserts T_i along with its lock requests into a waiting queue. Every time the scheduler releases the locks of a completed transaction, it examines the waiting queue to see if it can grant all of the lock requests