C. Shadow Paging. (Doesn’t require a log file)

The database is made up of some number of disk pages. A "page table" is constructed with one entry per data page (The page address).
When a Tx begins, the page table is copied to disk. This copy is called the "Shadow Page Table". When a data page is written upon, a completely new copy of the page is made containing the new/changed data. The address of the new page is entered on the current page table (in main memory)
If an error occurs, roll back consists entirely of replacing the current page table by the shadow page table.
Commit just means disposing of the shadow page table.
Advantages: VERY FAST, no log required.
Disadvantages: Page table can be large enough that maintaining it is costly
Data pages move! Sequential processing is affected.

Concurrency Control

Defn. Given a set of transactions \{T1,T2,...Tn\}, a schedule S of that set is an ordering of the various operations in the Tx's, such that for any individual Tx, the operations are performed in the correct sequence.

Defn. A schedule is called Serial if the operations of each individual Tx are performed sequentially.

Defn. A schedule is called "serializable" if it is equivalent to some serial schedule...

What does "equivalent" mean?

1st try: Result equivalence. S1 is equivalent to a serial schedule S, if, after S1 has occurred the database is in the same state as it would be if S had executed.

There is no way to check result equivalence without actually checking all possible serial schedules, and even if we could do that it might allow two schedules to be equivalent that really shouldn’t be...complicated example

T0: 5% raise rounded up to the nearest $10,000
T1: 1,000 raise
Assume that the current salary is $100,000

T0 followed by T1: 100000->105000->110000->111000
T1 followed by T0: 100000->101,000->106050->110000
Consider the following schedule S1

\[
\text{<Read, T0, Salary, 100,000>}
\]
\[
s:=1.05*s
\]
\[
\text{<Read, T1, Salary, 100,000>}
\]
\[
S := S + 1000
\]
\[
\text{<Write, T1, Salary, 100,000, 101,000>}
\]
\[
\text{Round(s)}
\]
\[
\text{<Write, T0, Salary, 100,000, 110,000>}
\]

The net value of salary will be 110,000 so it satisfies the definition of Result Serializability, but it is an accident....do this for salary being 10000 and T0 rounded up to 1000.

2\textsuperscript{nd} try: State equivalence. S1 is equivalent to a serial schedule S, if, after S1 has occurred the database is in the same state as it would be if S had executed, for any possible original database state.

This is very difficult to check. So there is a sufficient condition for a schedule to be serializable (the one used in most commercial databases), and it is called conflict-serializability. In this sense we supply the next definition.

The real definition (Conflict-serializability)

Schedules S1 and S2 are equivalent if they can be turned one into the other by a sequence of nonconflicting swaps of adjacent actions

Conflicts: Given two different transactions \(T_i\) and \(T_j\)

1) Read \((T_i, X)\) and Read \((T_j, Y)\) never conflict, even if \(X = Y\).

2) Read \((T_i, X)\) and Write \((T_j, Y)\) conflict if \(X = Y\). So do Read \((T_j, Y)\) and Write \((T_i, X)\), when \(X = Y\).

3) Write \((T_i, X)\) and Write \((T_j, Y)\) conflict if \(X = Y\). So do Write \((T_j, Y)\) and Write \((T_i, X)\) when \(X = Y\).

Investigate the previous example to see which kind of equivalence we are dealing with.

Defn A schedule is serializable if it equivalent to some serial schedule.

This second definition of serializability is strictly structural and can be checked merely by investigating the text of the two schedules.
Precedence Graphs and a Test for Conflict-Serializability.

Because the conflicting actions put some constraints on the order of transactions, we can claim that if the constraints are not contradictory we can find a conflict-equivalent serial schedule.

Defn. Given a schedule $S$ involving transactions $T_1$ and $T_2$ (among others), we say that $T_1$ takes precedence over $T_2$ if there are actions $A_1$ of $T_1$ and $A_2$ of $T_2$ such that:

1. $A_1$ is ahead of $A_2$ in $S$,
2. Both $A_1$ and $A_2$ involve the same database element, and
3. At least one of $A_1$ and $A_2$ is a write action.

Note that these are the conditions that keep us from swapping the order of $A_1$ and $A_2$.

Precedence graph: The nodes of the precedence graph are the transactions of a schedule $S$, labeled according to the name of the transactions. There is an arc from node $T_i$ to node $T_j$ if $T_i$ takes precedence over $T_j$.

Test for Conflict-Serializability:

Construct the precedence graph for the schedule $S$. If there are any cycles, the schedule $S$ is not conflict-serializable.

Example: Consider the schedule $S_1$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A,t)</td>
<td>t := t + 100</td>
<td></td>
</tr>
<tr>
<td>s := s + 25</td>
<td></td>
<td>Write(A,t)</td>
</tr>
<tr>
<td>Write(B,s)</td>
<td></td>
<td>Read(A,v)</td>
</tr>
<tr>
<td>Write(B,s)</td>
<td></td>
<td>v := v*1</td>
</tr>
<tr>
<td>Read(B,t)</td>
<td></td>
<td>Write(A,v)</td>
</tr>
<tr>
<td>t := t*1.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B,t)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Looking at the actions on the value $B$, we can easily justify that $T_1$ takes precedence over $T_2$. The actions on the value $A$, determine that $T_2$ takes precedence over $T_3$. 
This graph does not have cycles, so it is possible to convert $S_1$ into the schedule in which all the actions of each of the three transactions occur in a serial order:

$$
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
\text{Read}(B,s) & \text{Read}(A,t) & \text{Read}(A,v) \\
s := s + 25 & t := t + 100 & v := v \ast 1 \\
\text{Write}(B,s) & \text{Write}(A,t) & \text{Write}(A,v) \\
\end{array}
$$

On the other hand, if we consider the schedule $S_2$ with the same set of actions as before, but in different order:

$$
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
\text{Read}(B,s) & \text{Read}(A,t) & \text{Read}(A,v) \\
s := s + 25 & t := t + 100 & v := v \ast 1 \\
\text{Write}(B,s) & \text{Write}(A,t) & \text{Write}(A,v) \\
\end{array}
$$

Here again examining the actions on A we see that $T_2$ takes precedence over $T_3$. But when we examine B, we see that $T_1$ takes precedence over $T_2$ [because Read($T_1$, B) and
Write(T₁, B) appear before Write(T₂, B) but also T₂ takes precedence over T₁ [because Read(T₂, B) appears before Write(T₁, B)]. So the precedence graph for schedule S₂ would be

NOTE: Finding cycles has a very nice theoretical touch, but it does not work in real situations. You need to have the complete schedule to inspect before you can decide. Unfortunately the interleaving of operations is done in real time, and you don’t want to stop the process and start it all over again.

This means that in realistic situations it is not feasible to prepare a schedule and then test it for serializability...

What we have to do is to find a set of rules that we can impose on the way schedules are created so that we simply forbid non-serializable schedules.

Unfortunately...we will be forced to exclude some otherwise serializable schedules and therefore compromise efficiency.

Defn. A Lock is a variable associated to a data item that describes the availability of that item for access by a Tx's

1st try Binary Locks. A lock may assume only the values 1 (locked) and 0 (unlocked). If the lock for a particular data item is 1, the item cannot be accessed by a Tx.

The OS must support 2 new operations

Lock(X){
    if lock(X) == 0 lock(X) = 1
    else while ( lock(X) == 1 ) continue;
}

Whatever Tx that successfully executes Lock(X) is said to hold a lock on X.

Unlock(X){
    lock(X) = 0;
}

The Binary Locking Protocol...Tx's must adhere to the following rules:

1. A Tx must successfully execute Lock(X) before it executes either Read(X) or Write(X).
2. A TX must execute Unlock(X) after all reads and writes are finished
3. If a Tx already holds a lock on X, it cannot execute Lock(X).
4. If a TX does not hold a lock on X, it may not execute Unlock(X).

Example

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock(S)</td>
<td>Lock(S) //refused</td>
</tr>
<tr>
<td>Read(S)</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>Update S</td>
<td>Write(S)</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>Unlock(S)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

NOTE on implementation. The usual implementation of a locking scheme is via a separate "lock manager". Instead of a separate variable for every data item, the lock manager will maintain some kind of lock table which shows which Tx's hold locks on which data items.

If several processes may request a particular data item, the OS may maintain a "locking queue"

Lock(X) becomes

if queue(X) is empty, place Tx at the head of the queue
else place the Tx at the back of the queue.

Unlock(X) consists of deleting the head of the queue.

Multimode locking... a lock can have three values..

Read locked (a.k.a. Shared lock or S-Lock)
Write locked (a.k.a. Exclusive lock or X-Lock).
Unlocked,

The locking scheme is a shown in the box:

<table>
<thead>
<tr>
<th>Request \ Value</th>
<th>U</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td>Granted</td>
<td>Granted</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>Granted</td>
<td>Refused</td>
</tr>
</tbody>
</table>

The OS must support three primitive operations:
read_lock(X)
write_lock(X)
unlock(X)
whose implementation is similar to the operations for binary locks.

The multimode locking protocol...each Tx must follow the following set of rules:

1. A Tx must execute read_lock(X) or write_lock(X) before it executes read(X)
2. A Tx must execute write_lock(X) before it executes write(X)
3. A Tx must execute unlock(X) after all reads and writes have finished.
4. A Tx may not execute read_lock(X) if it already holds a read lock nor write_lock(X) if it holds a write lock.
5. A Tx may not execute unlock(X) unless it holds a lock on X.

Multimode locking does not guarantee serializability.

Example

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_lock(Y)</td>
<td>read_lock(X)</td>
</tr>
<tr>
<td>read(Y, y)</td>
<td>Read(X,y)</td>
</tr>
<tr>
<td>unlock(Y)</td>
<td>Unlock(X)</td>
</tr>
<tr>
<td>&lt;Time out&gt;</td>
<td>Write_lock(Y)</td>
</tr>
<tr>
<td></td>
<td>Read(Y,y)</td>
</tr>
<tr>
<td></td>
<td>Y = x+y</td>
</tr>
<tr>
<td></td>
<td>Write(Y,y)</td>
</tr>
<tr>
<td></td>
<td>Unlock(Y)</td>
</tr>
<tr>
<td>write_lock(X)</td>
<td>write_lock(X)</td>
</tr>
<tr>
<td>read(X,x)</td>
<td>read(X,x)</td>
</tr>
<tr>
<td>x = x+y</td>
<td>x = x+y</td>
</tr>
<tr>
<td>write (X,x)</td>
<td>Write (X,x)</td>
</tr>
<tr>
<td>Unlock(X)</td>
<td>Unlock(X)</td>
</tr>
</tbody>
</table>

Assume the initial values of X and Y are X = 20 and Y = 30
then this schedule results in x = 50 and y = 50, but
the schedule T0T1 yields x = 50, y = 80, and
the schedule T1T0 yields x = 70, y = 50

Also, using the precedence graph, we see that
Since Read (T0, Y) appears before Write (T1, Y), and similarly, T1 takes precedence over T0, because Read (T1, X) appears before Write (T0, X).

Finally!!

The two phase locking protocol (for either binary or multimode locking)

Each Tx must go through two phases. In the first phase, a Tx may acquire new locks but no locks may be released. In the second phase, locks may be released, but no new locks may be acquired

The net effect of TPLP is that once a Tx starts releasing locks, it can't acquire any new locks.

Thrm: The TPLP guarantees serializability
Pf: There are essentially 4 possible cases in which the test for serializability can be violated:

case 1.

\[
\begin{array}{c|c}
T0 & T1 \\
\\
\ldots & \ldots \\
read(X,x) & write(X,y) \\
write(Y,y) & read(Y,y) \\
\ldots & \ldots \\
\end{array}
\]

It's easy to show that this schedule cannot be made to satisfy the TPLP

other 3 cases... homework.

TPLP doesn't solve all the problems:

\[
\begin{array}{c|c}
T0 & T1 \\
\\
read\_lock(Y) & read\_lock(X) \\
read(Y) & read(X) \\
write\_lock(X) & write\_lock(Y) \\
\end{array}
\]

Defn. A situation in which a sequence of Tx's is waiting for another to finish in such a way that none can finish is called "deadlock"
Deadlock can be dealt with by
   1. Deadlock avoidance (don’t let it happen)
   2. Deadlock detection and recovery

1. Deadlock can be avoided by using a more restricted locking protocol.

   First try: A Tx must request and be granted all locks before it begins any
   processing...(Slows down concurrency a lot! )
   Second try. A prioritized locking system. Data items are (partially) ordered.

   The protocol is
   A Tx's first lock request can be for any data item.
   After that a Tx cannot request a lock on a data item unless it already holds
   a lock on that data item’s immediate superior.

Example

```
   A
   /\ \
  / \  \
 B   H
   / \  \
  /   \ \\ \\
 C     D   I
   / \  /  \\
  /   / \\
 E   F   G  K
```

(If you need both B & H, you first have to lock A)

The ordering can as simple as Tables -> pages-> records

This system works but requires a lot of overhead

2. Deadlock detection:

   a. The "wait-for" graph is a bipartite graph, one of whose node sets corresponds to
      Tx's the other to a set of resources (data items)...draw a line from Ti to Dj if Ti
      has attempted to execute lock(Dj)
Deadlock occurs if the wait for graph contains a cycle.

b. brute force. Keep track of which processes are in a wait state. If more than 2 processes are in wait state for some number of seconds, assume that deadlock has occurred.

In either case, Deadlock recovery means killing one or the other of the Tx's (causes a system error)

ASIDE... multimode can also cause another problem called "livelock"

```
T0       T1       T2
read_lock(X)  write_lock(X)  read_lock(X)
unlock(X)     (Refused)     write_lock(X)
```

Livelock can be avoided by

1. "Fair locking". All locking requests are queued. When a write_lock enters the queue, no more read_locks can be granted until the write_lock executes (read_locks must go behind the write lock on the queue). This works but can really screw up throughput
2. Priority queues and aging. All lock requests are given a priority..Read_locks are always given higher priority. When unlock is executed, the lock holding the highest priority is then granted. Periodically, all requests in the queue are "aged", ie have their priority increased so eventually every lock is granted.