Today’s lecture

- Discuss some recovery concepts
- Do some reviews
Failure Modes

- Data must be protected from corruption

- How can data get corrupted?
  - Erroneous Data Entry
    - caught through triggers, key, foreign-key, constraints
  - Media Failures
    - caught by parity checks on sectors
    - RAID
  - Catastrophic Failures
    - Distribute data
  - System Failures
    - Recovery algorithms
System Failures

- May cause the contents of main memory to be lost
- This causes state of the transaction to be lost
- How do we ensure the **atomicity** and **durability** properties?
- We record the database changes in a non-volatile storage called the *log*
- Recover from the log when a failure occurs
Address Spaces Associated with a Transaction

- The space of disk blocks holding data items
- The main memory buffer that holds the pages for read/write operations
- The local address space of the transaction
Primitives associated with a transaction

- **INPUT**(X): Copy the disk block containing the data item X to a memory buffer
- **READ**(X, t): Copy the data item X to the transaction’s local variable t
  - If the block containing the data item X is not in a memory buffer, then execute **INPUT**(X)
  - Copy the value of X to local variable t
- **WRITE**(X, t): Copy the value of local variable t to data item X in a memory buffer
  - If the block containing the data item X is not in a memory buffer, then execute **INPUT**(X)
  - Copy the value of t to X in the buffer
- **OUTPUT**(X): Copy the block containing X from its buffer to disk
Example

Consider the transaction: 

\[
\text{READ}(A, t); t := t \times 2; \text{WRITE}(A, t) \\
\text{READ}(A, t); t := t \times 2; \text{WRITE}(B, t)
\]

<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:= t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t := t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

What happens if there is a system failure just before OUTPUT(B)?
Log

- Create a *log* of all important actions
- A log is a sequence of *log records*
- Each log record says something that a transaction has done
  - **Intention**: If there is a crash, the log will tell us how to restore the consistency of the database
  - **Problems**:
    - If the log isn’t on the disk, it too can be lost.
    - If we have to write every log entry to disk, we do a lot of disk I/O
- Log records are initially written on main memory blocks
- Log records are transferred to the disk periodically
Undo (Write-Ahead) Logging

The following log records are stored for undo logging:

- `<START T>`: indicates that transaction $T$ has begun
- `<COMMIT T>`: transaction $T$ has completed successfully
- `<ABORT T>`: transaction $T$ could not complete successfully
- `<T, X, v>`: transaction $T$ has changed $X$ and its former value is $v$
Undo Logging Rules

- Create a log record for every operation
- If transaction $T$ modifies data item $X$, then log record $< T, X, v >$ must be written before the new value of $X$ is written
- If a transaction commits, then its $COMMIT$ log record must be written to disk only after all data items changed by the transaction have been written to disk, but as soon thereafter as possible
Order of writing to disk

Materials associated with a transaction must be written to disk in the following order

1. The log records indicating changed data items
2. The changed data items themselves
3. The *COMMIT* log record

The order of (1) and (2) applies to each data item individually, not to the group of update records for a transaction as a whole. Log records need to be forced into the disk. We refer to this as *flushing* the log.
Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>READ(A,t)</strong></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt; START T &gt;</td>
</tr>
<tr>
<td>( t := t \times 2 )</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>WRITE(A,t)</strong></td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt; T,A,8 &gt;</td>
</tr>
<tr>
<td><strong>READ(B,t)</strong></td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t := t \times 2 )</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>WRITE(B,t)</strong></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt; T,B,8 &gt;</td>
</tr>
<tr>
<td><strong>FLUSH LOG</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OUTPUT(A)</strong></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>OUTPUT(B)</strong></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td><strong>FLUSH LOG</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt; COMMIT T &gt;</td>
</tr>
</tbody>
</table>
Recovery Using Undo Logging

The recovery manager looks at the log to decide if a transaction $T$ is committed or not.

- If there is a log record $<\text{COMMIT } T>$, then by undo rule all changes made by $T$ were written to the disk.
- If we find a $<\text{START } T>$ but no $<\text{COMMIT } T>$, $T$ is an incomplete transaction.
  - Maybe some changes made by $T$ were written on disk, and some were not.
  - All changes made by $T$ must be undone.
  - If $T$ changed $X$ on the disk, then by undo rule $<T,X,v>$ record must be on the log.
Recovery Algorithm

- We must be systematic about the order in which we undo transactions.
- Recovery manager scans the log from the end.
  - If $T$ is a transaction whose $COMMIT$ record has been scanned, then do nothing.
  - If $T$ is an incomplete transaction or an aborted transaction, then:
    - for each record $<T,X,v>$ change the value of $X$ to $v$
    - write a log record $<ABORT\ T>$ and flush the log.
## Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt; START T &gt;</td>
</tr>
<tr>
<td>2</td>
<td>READ(A, t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t := t * 2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>WRITE(A, t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt; T,A,8 &gt;</td>
</tr>
<tr>
<td>5</td>
<td>READ(B, t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>t := t * 2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>WRITE(B, t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt; T,B,8 &gt;</td>
</tr>
<tr>
<td>8</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt; COMMIT T &gt;</td>
</tr>
<tr>
<td>12</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scenarios

1. Crash occurs after step 12
   - `<COMMIT T>` record in disk
   - All log records concerning `T` are ignored during recovery

2. Crash occurs between steps 11 and 12
   - If `<COMMIT T>` in disk, then exactly like previous scenario 1
   - If `<COMMIT T>` not in disk, then `T` is considered incomplete and the following operations are done in order
     (a) It gets record `<T, B, 8>` and changes value of `B` to 8
     (b) It gets record `<T, A, 8>` and changes value of `A` to 8
     (c) `<ABORT T>` is written to the log, and log is flushed
3. Crash occurs between steps 10 and 11
   - COMMIT record was not written so $T$ is undone as in scenario 2

4. Crash occurs between steps 8 and 10
   - $T$ is undone as in scenario 3

5. Crash occurs before step 8
   - We do not know whether any log record pertaining to $T$ reached the disk
   - If the change to $A$ or $B$ reached the disk, then the corresponding log record reached the disk, and the change can be undone during recovery
Checkpointing

- Recovery as described, requires us to examine the entire log.
- Log files are large and scanning the entire log file may be expensive.
- To avoid this problem, we *checkpoint* the log periodically.
Simple Checkpointing Algorithm

- The algorithm
  - Stop accepting new transactions
  - Wait until all active transactions commit or abort and have written a *COMMIT* or *ABORT* record on the log
  - Flush the log to the disk
  - Write a log record *< CKPT >* and flush the log again
  - Resume accepting new transactions
- There are no incomplete transactions prior to *< CKPT >*
- So during recovery scan backwards until we encounter *< CKPT >*
Example

Suppose the log begins:

\(< \text{START} \ T_1 >\)
\(< T_1, A, 5 >\)
\(< \text{START} \ T_2 >\)
\(< T_2, B, 10 >\)

At this point we decide to do a checkpoint

We have to wait for $T_1$, $T_2$ to complete before we can checkpoint
Example Contd.

Suppose the continuation of the log is as follows:

\(< \text{START } T_1 >\)
\(< T_1, A, 5 >\)
\(< \text{START } T_2 >\)
\(< T_2, B, 10 >\)
\(< T_2, C, 15 >\)
\(< T_1, D, 20 >\)
\(< \text{COMMIT } T_1 >\)
\(< \text{COMMIT } T_2 >\)
\(< \text{CKPT} >\)
\(< \text{START } T_3 >\)
\(< T_3, E, 25 >\)
\(< T_3, F, 30 >\)

At this point a crash occurs
Nonquiescent Checkpointing

Problem with simple checkpointing is that no new transactions are accepted while checkpointing is being done.

This problem is avoided with nonquiescent checkpointing:

1. Write a log record \(< \textit{START CKPT} (T_1, \ldots, T_k) >\) where \(T_1, \ldots, T_k\) are active transactions and flush the log.
2. Wait until all of \(T_1, \ldots, T_k\) commit or abort, but do not prohibit other transactions from starting.
3. When all of \(T_1, \ldots, T_k\) have completed, write a log record \(< \textit{END CKPT} >\) and flush the log.
Recovery with Nonquiescent Checkpointing

- Scan the log from the end to find all incomplete transactions and undo them.

- While scanning backwards we may first encounter an \(< END CKPT >\) or \(< START CKPT(T_1, \ldots, T_k) >\).

- If we first encounter \(< START CKPT(T_1, \ldots, T_k) >\), then the only incomplete transactions are those that we met before encountering \(< START CKPT(T_1, \ldots, T_k) >\) and those of \(T_1, \ldots, T_k\) that did not complete before the crash. We need not scan further back than the start of the earliest of these incomplete transactions.

- If we first encounter \(< END CKPT >\), then we know that all incomplete transactions began after the previous \(< START CKPT(T_1, \ldots, T_k) >\) record. We may scan backwards as far as the \(START CKPT\) and then stop.
Consider the following log:

\(< \text{START } T_1 >\)
\(< T_1, A, 5 >\)
\(< \text{START } T_2 >\)
\(< T_2, B, 10 >\)

At this point we decide to do a checkpoint. We insert \(< \text{START CKPT}(T_1, T_2) >\)
Example

Suppose the continuation of the log is as follows:

\(< \text{START } T_1 >\)
\(< T_1, A, 5 >\)
\(< \text{START } T_2 >\)
\(< T_2, B, 10 >\)
\(< \text{START CKPT}(T_1, T_2) >\)
\(< T_2, C, 15 >\)
\(< \text{START } T_3 >\)
\(< T_1, D, 20 >\)
\(< \text{COMMIT } T_1 >\)
\(< T_3, E, 25 >\)
\(< \text{COMMIT } T_2 >\)
\(< \text{END CKPT} >\)
\(< T_3, F, 30 >\)

At this point a crash occurs
Example

Suppose the crash occurred during checkpointing as follows:

\(< \text{START } T_1 >\)
\(< T_1, A, 5 >\)
\(< \text{START } T_2 >\)
\(< T_2, B, 10 >\)
\(< \text{START CKPT}(T_1, T_2) >\)
\(< T_2, C, 15 >\)
\(< \text{START } T_3 >\)
\(< T_1, D, 20 >\)
\(< \text{COMMIT } T_1 >\)
\(< T_3, E, 25 >\)

At this point a crash occurs
Problems with Undo Logging

- Has a large number of disk I/Os
- Performance is an issue
Undo Logging vs. Redo Logging

- Undo logging cancels the effect of incomplete transactions, ignores committed transactions during recovery.
- Redo logging ignores incomplete transactions, and repeats the changes made by committed transactions.
- Undo logging requires us to write changed database items to disk before the COMMIT log record reaches the disk.
- Redo logging requires that the COMMIT record appear on the disk before any changed values reach disk.
- Undo logging requires you to store the old value of data items.
- Redo logging requires you to store the new value of data items.
Log Records in Redo Logging

- \(< \text{START } T >\): indicates that transaction \( T \) has begun
- \(< \text{COMMIT } T >\): transaction \( T \) has completed successfully
- \(< \text{ABORT } T >\): transaction \( T \) could not complete successfully
- \(< T, X, v >\): transaction \( T \) has changed \( X \) and its new value is \( v \)
Redo-Logging Rule

Before modifying any data item $X$ on disk, it is necessary that all log records pertaining to this modification of $X$, including both the update record $< T, X, v >$ and the $< COMMIT T >$ record, must appear on disk.
Order of writing to disk

1. The log records indicating changed data items
2. The \textit{COMMIT} log record
3. The changed data items themselves
## Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt; START T &gt;</td>
</tr>
<tr>
<td>t := t * 2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt; T,A,16 &gt;</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t := t * 2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt; T,B,16 &gt;</td>
</tr>
<tr>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt; COMMIT T &gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Observations about Redo Logging

- If the log does not have a `<COMMIT T>` record, then no changes made by $T$ are written on the disk.

- If the log has a `<COMMIT T>` record, then changes made by $T$ may or may not be written on the disk.
Recovery with Redo Logging

1. Identify the **committed transactions**

2. Scan the log forward from the beginning. For each log record \(< T, X, v >\) do
   (a) If \( T \) is not a committed transaction, do nothing
   (b) If \( T \) is committed, write value \( v \) for data item \( X \)

3. For each incomplete transaction \( T \), write an \(< ABORT \ T >\) record to the log and flush the log
### Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt; START T &gt;</td>
</tr>
<tr>
<td>2</td>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t := t * 2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt; T,A,16 &gt;</td>
</tr>
<tr>
<td>5</td>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>t := t * 2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt; T,B,16 &gt;</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt; COMMIT T &gt;</td>
</tr>
<tr>
<td>9</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Scenarios

1. Crash occurs after Step 9
   - `<COMMIT T>` has been flushed to disk, so $T$ is considered a committed transaction
   - The recovery manager writes value 16 for $A$ and then writes 16 for $B$

2. Crash occurs between steps 8 and 9. There can be two cases:
   - `<COMMIT T>` got to the disk – recovery same as scenario 1
   - `<COMMIT T>` did not get to the disk – recovery same as scenario 3

3. Crash occurs prior to step 8
   - `<COMMIT T>` did not get to the disk, $T$ is considered incomplete
   - No actions taken by the recovery manager for $T$
Checkpointing with Redo Logging

- Write a log record \(< \text{START CKPT} (T_1, \ldots, T_k) >\) where \(T_1, \ldots, T_k\) are all the active transactions and flush the log.
- Write to disk all data items that were written to buffers but not yet to disk by transactions that had already committed when the \(< \text{START CKPT} >\) record was written to the log.
- Write an \(< \text{END CKPT} >\) record to the log and flush the log.
Example

\[< \text{START } T_1 >\]
\[< T_1, A, 5 >\]
\[< \text{START } T_2 >\]
\[< \text{COMMIT } T_1 >\]
\[< T_2, B, 10 >\]
\[< \text{START CKPT}(T_2) >\]
\[< T_2, C, 15 >\]
\[< \text{START } T_3 >\]
\[< T_3, D, 20 >\]
\[< \text{END CKPT} >\]
\[< \text{COMMIT } T_2 >\]
\[< \text{COMMIT } T_3 >\]
Recovery with a Checkpointed Redo Log

If the last checkpoint record on the log before a crash is `<END CKPT>`

- every value written by transactions that were committed before the corresponding `<START CKPT(T_1, ..., T_k)>` has its changes written onto the disk, and we need not concern ourselves with them

- transactions in $T_1, ..., T_k$ or transactions that started after the beginning of the checkpoint can still have changes that have not yet migrated to the disk even though the transaction has committed. These transactions must be redone

- In searching the log we have to scan as far back as the earliest of `<START T_i>`
If the last checkpoint record on the log before a crash is
\( <\text{START CKPT}(T_1, \ldots, T_k) > \)

- We cannot be sure that committed transactions prior to the
  start of this checkpoint had their changes written to disk.

- We scan back to the previous \( <\text{END CKPT} > \) record and
  find its matching
  \( <\text{START CKPT}(S_1, \ldots, S_m) > \) record, and redo all those
  committed transactions that either started after
  \( <\text{START CKPT}(S_1, \ldots, S_m) > \) or are among the \( S_i \)’s

- In searching the log records we have to scan back to the
  earliest of \( <\text{START } S_i > \)
Example 1

\[ \langle \text{START } T_1 \rangle \]
\[ \langle T_1, A, 5 \rangle \]
\[ \langle \text{START } T_2 \rangle \]
\[ \langle \text{COMMIT } T_1 \rangle \]
\[ \langle T_2, B, 10 \rangle \]
\[ \langle \text{START CKPT}(T_2) \rangle \]
\[ \langle T_2, C, 15 \rangle \]
\[ \langle \text{START } T_3 \rangle \]
\[ \langle T_3, D, 20 \rangle \]
\[ \langle \text{END CKPT} \rangle \]
\[ \langle \text{COMMIT } T_2 \rangle \]
\[ \langle \text{COMMIT } T_3 \rangle \]

At this point crash occurs
Example 2

\(< START \, T_1 >\)
\(< T_1,A,5 >\)
\(< START \, T_2 >\)
\(< COMMIT \, T_1 >\)
\(< T_2,B,10 >\)
\(< START \, CKPT(T_2) >\)
\(< T_2,C,15 >\)
\(< START \, T_3 >\)
\(< T_3,D,20 >\)
\(< END \, CKPT >\)
\(< COMMIT \, T_2 >\)

At this point crash occurs
Example 3

<START T₁>
<T₁, A, 5>
<START T₂>
<COMMIT T₁>
<T₂, B, 10>
<START CKPT(T₂)>
<T₂, C, 15>
<START T₃>
<T₃, D, 20>

At this point crash occurs
Lecture Objectives

Today’s lecture
- Completed discussion on recovery
- Do some reviews