Comparison of Undo and Redo Logging:

- **Undo logging**: data must be written to disk immediately after a transaction finishes; potentially increases the number of I/O operations required
- **Redo logging**: modified blocks must be kept in buffers until the transaction commits and log has been flushed; potentially increases the number of buffers required by transactions
- Both may impose contradictory requirements during checkpointing
- **Undo/Redo logging** is more flexible than Undo or Redo logging
  - … but is also more costly
Undo/Redo Logging

- Undo/Redo log records differ only in the update records: 
  \( <T,X,v,w> \) means that transaction T changed the value of DB element X from v to w

- Rule: before modifying any DB element X on disk, the update record \( <T,X,v,w> \) must appear on disk
  - \( <\text{COMMIT } T> \) record may precede or follow any of the changes to the DB elements on disk
  - more liberal than Undo or Redo logging

- For recovery, it permits either restoring the DB state or repeating the changes made. Policy:
  - Redo committed transactions (earliest first)
  - Undo incomplete transactions (latest first)
  - Both are necessary to avoid incomplete recovery
Recovery with Undo/Redo Logging

Example:
Read (A,t);  t ← t \times 2
Write (A,t);
Read (B,t);  t ← t \times 2
Write (B,t);
FLUSH LOG;
Output (A);
Output (B);

<START T>

If failure occurs after <COMMIT T> has been written on disk, T is treated as committed and the value 16 is written on disk for A and B

<T,A,8,16>

If failure occurs before <COMMIT T> appears on disk, T is treated as an incomplete transaction and the value 8 is written on disk for A and B

<T,B,8,16>

<COMMIT T>
Recovery with Undo/Redo Logging

- An additional rule may be used in order to avoid situations where, due to delayed commitment, a transaction appears (to the user) to have committed, but the <COMMIT > record has not been written on disk
  - a crash would cause such a transaction to be undone
  - **Rule**: A <COMMIT T> record must be flushed to disk as soon as it appears in the log

- **Order** of redo and undo during recovery:
  - doesn’t really matter
  - we still cannot prevent a committed transaction that must be redone to read a value written by an incomplete transaction that must be undone (**dirty read**)
  - need to **isolate** such transactions (**concurrency control**)

Lecture 2
Nonquiescent checkpointing:
1. Write \texttt{<START CKPT(T1,T2,\ldots,Tk)>} to log for active transactions T1,T2,\ldots,Tk and flush the log
2. Write to disk \textbf{all} dirty buffers (those that contain changed DB elements)
3. Write \texttt{<END CKPT>} and flush the log

Step 2 enforces the writing on disk of elements changed by incomplete transactions

The following constraint must be observed though:
- A transaction may not write any values until it is certain not to abort
Checkpointing with Undo/Redo Logging

Example:

During the checkpoint, A and B will be flushed to disk.

if failure occurs at the end, T2 and T3 are considered complete and are redone; T1 is considered to be complete and to have had its changes written on disk

if failure occurs before <COMMIT T3>, then T2 is considered complete whereas T3 is incomplete; redo T2 by writing 15 for C on disk; no need to write 10 for B; undo T3 by writing 19 for D; if T3 were active at the start of the checkpoint, we would need to check if other changes by T3 need to be undone
Concurrence Control

I.e., how to maintain the DB in a consistent state in the presence of constraints when multiple transactions execute concurrently

Example:

T1: Read(A)
    A ← A+100
    Write(A)
    Read(B)
    B ← B+100
    Write(B)

T2: Read(A)
    A ← A×2
    Write(A)
    Read(B)
    B ← B×2
    Write(B)

Constraint: A=B
## Schedules

<table>
<thead>
<tr>
<th>Schedule A</th>
<th></th>
<th>T1</th>
<th></th>
<th>T2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Read(A); A ← A+100</td>
<td>Write(A);</td>
<td>Read(B); B ← B+100;</td>
<td>Write(B);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write(A);</td>
<td></td>
<td>Read(A);A ← A×2;</td>
<td>Write(A);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read(B);</td>
<td></td>
<td>Read(B);B ← B×2;</td>
<td>Write(B);</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>25</td>
<td></td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>25</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>
## Schedules

<table>
<thead>
<tr>
<th>Schedule B</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>T2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2; Write(A); Read(B); B ← B×2; Write(B);</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Read(A); A ← A+100; Write(A); Read(B); B ← B+100; Write(B);</td>
<td>150</td>
<td>150</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>
## Schedules

<table>
<thead>
<tr>
<th>Schedule C</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A); A ← A+100</td>
<td>Read(A); A ← A×2;</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td>Write(A);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B+100;</td>
<td>Read(B); B ← B×2;</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td>Write(B);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>25</td>
<td></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>125</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>250</td>
<td></td>
<td>250</td>
<td></td>
</tr>
</tbody>
</table>
## Schedules

<table>
<thead>
<tr>
<th>Schedule D</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Read(A); A ← A+100</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>T2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>150</td>
</tr>
</tbody>
</table>
## Schedules

<table>
<thead>
<tr>
<th>Schedule E</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Read(A); A ← A+100</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Write(A);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>T2'</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Read(A); A ← A×1;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Write(A);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Read(B); B ← B×1;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Write(B);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Read(B); B ← B+100;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Write(B);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Schedule E</strong></td>
<td><code>A</code></td>
<td><code>B</code></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td><code>Read(A); A ← A+100</code></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td><code>Write(A);</code></td>
<td></td>
<td>125</td>
</tr>
<tr>
<td><code>Read(B); B ← B×1;</code></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td><code>Write(B);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Read(B); B ← B+100;</code></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td><code>Write(B);</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Schedule E</strong></th>
<th><code>A</code></th>
<th><code>B</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Read(A); A ← A+100</code></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td><code>Write(A);</code></td>
<td></td>
<td>125</td>
</tr>
<tr>
<td><code>Read(B); B ← B×1;</code></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td><code>Write(B);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Read(B); B ← B+100;</code></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td><code>Write(B);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Schedule E</strong></td>
<td><code>A</code></td>
<td><code>B</code></td>
</tr>
<tr>
<td><code>Read(A); A ← A+100</code></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td><code>Write(A);</code></td>
<td></td>
<td>125</td>
</tr>
<tr>
<td><code>Read(B); B ← B×1;</code></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td><code>Write(B);</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Read(B); B ← B+100;</code></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td><code>Write(B);</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Transaction Scheduling

- Only want to allow the execution of “good” schedules regardless of
  - initial states
  - transaction semantics
- Only the order of reads and writes matters
- **Example**: given a schedule of the form

  \[\text{Sc=r}_1(A)\text{w}_1(A)\text{r}_2(A)\text{w}_2(A)\text{r}_1(B)\text{w}_1(B)\text{r}_2(B)\text{w}_2(B)\]

  determine whether it should be allowed to proceed without causing inconsistency problems
Transaction Scheduling

- Scheduler must chose a “correct” interleaving of transaction operations among all possible interleavings.
- Recall that transactions should appear to execute in isolation.
- Hence, a “correct” interleaving should behave as if the transactions involved in the schedule were executed in isolation.

Example:

\[ Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

\[ Sc' = r_1(A)w_1(A) r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B) \]

Are the two equivalent?
Transaction Scheduling

**Example:** Which is the equivalent serial schedule of:

\[ S_d = r_1(A)w_1(A)r_2(A)w_2(A) \ r_2(B)w_2(B)r_1(B)w_1(B) \]

T1 must precede T2 in any equivalent schedule
T2 must precede T1 in any equivalent schedule

So \( S_d \) cannot be rearranged into a serial schedule, hence \( S_d \) is not equivalent to any serial schedule, hence \( S_d \) is “bad”

But, \( S_c = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \) is good

\[ T_1 \rightarrow T_2 \]  \[ T_1 \rightarrow T_2 \]
Transaction Concepts

- **Transaction**: sequence of \( ri(x) \), \( wi(x) \) actions
- **Conflicting actions**: \[ r1(A) \quad w2(A) \quad w1(A) \]
  \[ w2(A) \quad r1(A) \quad w2(A) \]
- **Schedule**: represents chronological order in which actions are executed
- **Serial schedule**: no interleaving of transactions
- What about concurrent schedules?
  - if they don’t involve conflicting actions, then any low-level synchronization mechanism is sufficient for scheduling
Definitions

- Schedules $S_1$, $S_2$ are **conflict equivalent** if $S_1$ can be transformed into $S_2$ by a series of swaps on non-conflicting actions.

- A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule.

- For a schedule $S$, its **precedence graph** $P(S)$ is defined as follows:
  - **Nodes**: transactions in $S$
  - **Arcs**: $T_i \rightarrow T_j$ whenever
    - $pi(A)$, $qj(A)$ are actions in $S$
    - $pi(A) \prec_S qj(A)$
    - at least one of $pi$, $qj$ is a write
Results

- **Lemma** S1, S2 conflict equivalent $\Rightarrow$ P(S1)=P(S2)
- **Proof**: Assume P(S1) $\neq$ P(S2)
  
  Then, $\exists$ Ti: Ti $\rightarrow$ Tj in S1 and not in S2
  
  S1 = $\ldots$pi(A)... qj(A)...
  pi, qj
  
  S2 = $\ldots$qj(A)...pi(A)...
  
  conflict

  Hence, S1 and S2 cannot be conflict equivalent

- **Note**: P(S1)=P(S2) does not imply S1, S2 conflict equivalent

- **Counterexample**:
  
  S1=w1(A) r2(A) w2(B) r1(B)
  
  S2=r2(A) w1(A) r1(B) w2(B)
Results

- **Theorem:** $P(S_1)$ acyclic iff $S_1$ conflict serializable

- **Proof:**
  - (if) Assume $S_1$ is conflict serializable
    - $\exists S_s: S_s, S_1$ conflict equivalent
    - $P(S_s) = P(S_1)$
    - $P(S_1)$ acyclic since $P(S_s)$ is acyclic
  - (only if) Assume $P(S_1)$ is acyclic

  Transform $S_1$ as follows:
  1. Take $T_1$ to be transaction with no incoming arcs
  2. Move all $T_1$ actions to the front
  3. We now have $S_1 = < T_1 \text{ actions } > < \text{... rest ...} >; \text{ remove } T_1 \text{ and incident arcs}$
  4. Repeat above steps to serialize rest!
Enforcing Serializability

- Option 1: allow any schedule; check for cycles in precedence graph (reactive)
- Option 2: prevent cycles in precedence graphs of schedules (proactive)
- Locking protocols are used to implement Option 2
  - New actions: lock (exclusive), unlock
  - Scheduler maintains information about locks in a lock table
- Rule 1: well-formed transactions
  - A transaction is well-formed when every operation on a database item X is preceded by a lock request on X and followed by an unlock request on X
- Rule 2: legal schedule
  - A schedule is legal if no lock request is granted to a transaction Tj for a database item X when a transaction Ti has already been granted the lock to X
Concurrency Control

- **2-Phase Locking**: all lock requests precede all unlock requests
  - Idea: $T_i = \ldots \text{li}(A) \ldots \ldots \text{ui}(A) \ldots$

---

# locks

held by

$T_i$

---

Growing

Shrinking

no unlock

no locks

---

time

Phase

Phase
2-Phase Locking

- Intuitively, each 2PL transaction may be thought to execute in its entirety at the moment it releases the first item.
- The conflict-equivalent serial schedule for a schedule $S$ of 2PL transactions is the one in which transactions are ordered in the same order as their first unlocks.
- **Conversion**: by induction on the number $n$ of transactions in a legal schedule $S$.
  - **Note**: conversion requires swapping the order of read and write operations of different transactions; while that is done, locks and unlocks can be ignored; once the actions are arranged serially, lock / unlock operations can be added.
Conversion by Induction

- Base case: \( n=1 \); \( S \) is already a serial schedule
- Induction:

Let \( S \) involve transactions \( T_1, T_2, \ldots, T_n \); let \( T_i \) be the first transaction to unlock an item by op. \( U_i(X) \). Then, it is possible to move all read/write actions of \( T_i \) to the beginning of \( S \) without passing any conflicting action.

Let \( T_i \) include an action \( W_i(Y) \). If there existed an action \( W_j(Y) \) in \( S \) that precedes \( W_i(Y) \), then \( U_j(Y) \) and \( L_i(Y) \) must also appear between \( W_j(Y) \) and \( W_i(Y) \).

We assumed that \( U_i(X) \) is the first unlock, hence it precedes \( U_j(Y) \). This means that \( U_i(X) \) must also appear before \( L_i(Y) \). But then, \( T_i \) is not a 2PL transaction.
2PL and Deadlocks

- 2PL cannot prevent deadlocks
- **Example**: consider the following schedule:
  L1(A), R1(A), L2(B), R2(B), W1(A), W2(B), L1(B), L2(A)

Each transaction waits for the other to release a lock
Beyond Simple 2PL

- Improvements to 2PL’s performance for allowing more concurrency:
  - shared locks
  - multiple granularities
  - inserts, deletes and phantoms
  - other types of concurrency control mechanisms

- Shared Locks
  - so far, locks have been of a single type: exclusive
  - read operations do not conflict
  - no need to lock exclusively for read
  - \( \text{SLi}(X) \): “Ti requests a shared lock on \( X \)”
  - \( \text{XLi}(X) \): “Ti requests an exclusive lock on \( X \)”
  - \( \text{Ui}(X) \): “Ti releases lock on \( X \)”
Requirements

- **Consistency**
  - An action $R_i(X)$ must be preceded by $S_{Li}(X)$ or $X_{Li}(X)$ with no intervening $U_i(X)$
  - An action $W_i(X)$ must be preceded by $X_{Li}(X)$ with no intervening $U_i(X)$
  - All locks must be followed by an unlock of the same element

- **2PL**
  - For any 2PL transaction $T_i$, no $S_{Li}(X)$ or $X_{Li}(X)$ can be preceded by $U_i(X)$

- **Legality**
  - If $X_{Li}(X)$ appears in a schedule, then there cannot be a following $X_{Lj}(X)$ or $S_{Lj}(X)$ for $j \neq i$ without an intervening $U_i(X)$
  - If $S_{Li}(X)$ appears in a schedule, then there cannot be a following $X_{Lj}(X)$ for $j \neq i$ without an intervening $U_i(X)$
Example

- Consider the following 2PL transactions
  
  T1: SL1(A), R1(A), XL1(B), R1(B), W1(B), U1(A), U1(B)
  T2: SL2(A), R2(A), SL2(B), R2(B), U2(A), U2(B)

- A legal interleaved execution of T1, T2 is as follows:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL1(A), R1(A)</td>
<td>SL2(A), R2(A), SL2(B), R2(B)</td>
</tr>
<tr>
<td>XL1(B) wait</td>
<td>XL1(B), R1(B), W1(B)</td>
</tr>
<tr>
<td></td>
<td>U2(A), U2(B)</td>
</tr>
<tr>
<td>U1(A), U1(B)</td>
<td></td>
</tr>
</tbody>
</table>
Upgrading Locks

- A transaction that wants to read and write an item, may first obtain a shared lock on the item for reading and then upgrade it to an exclusive lock for writing.

- Example:
  T1: SL1(A), R1(A), SL1(B), R1(B), XL1(B), W1(B), U1(A), U1(B)
  T2: SL2(A), R2(A), SL2(B), R2(B), U2(A), U2(B)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL1(A), R1(A)</td>
<td></td>
</tr>
<tr>
<td>SL1(B), R1(B), XL1(B)</td>
<td>SL2(A), R2(A), SL2(B), R2(B) wait</td>
</tr>
<tr>
<td>XL1(B) wait</td>
<td>U2(A), U2(B)</td>
</tr>
<tr>
<td>XL1(B), W1(B), U1(A), U1(B)</td>
<td></td>
</tr>
</tbody>
</table>
Upgrading Locks

- Lock upgrading may lead to deadlocks
- Example:

  T1                               T2

  SL1(A), R1(A)                    SL2(A), R2(A)
  XL1(A) wait                      XL2(A) wait
Update Locks

- We can avoid deadlocks caused by lock upgrade by using **update** locks:
  - An update lock $UL_i(X)$ allows transaction $T_i$ to read $X$ but not to write $X$
  - Only an update lock can be upgraded to a write lock
  - An update lock can be granted on $X$ when there are already shared locks on $X$
  - Once there is an update lock on $X$, no additional locks are allowed on $X$ (otherwise such a lock would never be upgraded to exclusive)
Example

T1: UL1(A), R1(A), XL1(A), W1(A), U1(A)
T2: UL2(A), R2(A), XL2(A), W2(A), U2(A)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>UL1(A), R1(A)</td>
<td>UL2(A) wait</td>
</tr>
<tr>
<td>XL1(A), W1(A), U1(A)</td>
<td>UL2(A), R2(A), XL2(A), W2(A), U2(A)</td>
</tr>
</tbody>
</table>
Increment Locks

- Several transactions operate on DB items by simply adding or subtracting constants
  - E.g., money transfer between accounts, seat reservations
- Such transactions commute with each other and their relative order doesn’t matter
- However, they don’t commute with transactions that read or write
- Assume transactions may include operations of the form INC(A,c), meaning that constant c is to be added to DB element A
  - INC(A,c) stands for: Read(A,t); t:=t+c; Write(A,t);
  - Increment actions need increment locks: ILi(X)
Increment Locks

- Increment locks do not enable reads or writes

Requirements:
- A consistent transaction can perform INCi(X) only if it is preceded by ILi(X)
- In a legal schedule, any number of transactions can hold an increment lock on item X at a time. If a transaction has an increment lock on X, no other transaction can have a shared or exclusive lock on X at the same time.
- INCi(X) conflicts with Rj(X) and Wj(X) for j<>i
- INCi(X) does not conflict with INCj(X)
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL1(A), R1(A), IL1(B), INC1(B), U1(A), U1(B)</td>
<td>SL2(A), R2(A), IL2(B), INC2(B), U2(A), U2(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL1(A), R1(A)</td>
<td>SL2(A), R2(A), IL2(B), INC2(B)</td>
</tr>
<tr>
<td>IL1(B), INC1(B)</td>
<td>U2(A), U2(B)</td>
</tr>
<tr>
<td>U1(A), U1(B)</td>
<td>U2(A), U2(B)</td>
</tr>
</tbody>
</table>
Granularity Issues

- Locking works well, but should we lock small or large objects?
  - If large objects (e.g., relations) are locked
    - Need few locks
    - Low concurrency
  - If small objects (e.g., tuples, attributes) are locked
    - Need more locks
    - More concurrency
- We can do both.
  - The bathroom metaphor:

```
<table>
<thead>
<tr>
<th>Stall 1</th>
<th>Stall 2</th>
<th>Stall 3</th>
<th>Stall 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>restroom</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

hall