Discuss how processes can synchronize
For example, agree on the ordering of events, or avoid accessing a shared resource simultaneously

Topics to be covered
Clock Synchronization
Logical Clocks
Global State
Election Algorithms
Mutual Exclusion
Distributed Transactions

Model
Assume we have N processes \( p_i \) (\( i = 1, 2, \ldots, N \))
Each process
- executes on a single processor
- has a state that changes as it executes
- executes a series of actions (either a message send or receive operation or an internal operation of the process (e.g., update of one of its variables))

Event: the occurrence of a single action
Events within a single process \( p_i \) can be placed in a single total order \(-\). Each process is characterized by its history, a series of events that occur at each process.
\[ h_i = \{ e_{i0}, e_{i1}, e_{i2}, \ldots \} \]
\( e_{i0} \): initial state

Clock Synchronization
When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Example (make)
<table>
<thead>
<tr>
<th>Computer on which compiler runs</th>
<th>2144</th>
<th>2145</th>
<th>2146</th>
<th>2147</th>
<th>Time according to local clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output c created</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computer on which editor runs</th>
<th>2142</th>
<th>2143</th>
<th>2144</th>
<th>2145</th>
<th>Time according to local clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output c created</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
How time is actually measured: Astronomically

- Transit of the sun: sun reaching its highest apparent point in the sky
- Solar day: interval between two consecutive sun transits. Solar second = 1/86400 of a solar day
- However, the period of the earth's rotation is not constant, earth is slowing down -> mean solar second

How time is actually measured: Atomic Time:

- Counting transitions of the cesium 133 atom

Based on the number of transitions per second of the cesium 133 atom (1 sec = 9,192,631,770 transitions)

Atoms are used to create cesium clocks, which are accurate to within a few picoseconds per day.

International Atomic Time (TAI)

- TAI seconds are of constant length, unlike solar seconds.
- UTC = TAI with leap seconds
- Introduce leap seconds whenever the discrepancy grows to 800 msec

UTC is broadcasted through short wave radio (WWV receivers) and satellite.

Satellites can give an accuracy of about ±0.5 ms

Is it possible to synchronize all clocks in a distributed system?

Each machine has a timer that causes an interrupt M times per second. A (software) clock keeps track of the number of ticks (interrupts) since some agreed-upon time in the past.

When the timer goes off, the interrupt handler adds 1 to the software clock

Let \( C \) be the value of the clock. Specifically, if UTC time is \( t \), let the value of the clock on machine \( p \) be \( C_p(t) \)

Perfect world, \( C_p(t) = t \) for all \( p \) and \( t \), \( dC_p/dt = 1 \) (i.e., \( C_p(t) = C_p(t') + dt \), \( \Delta t = t' - t \))

Theoretically, a timer with \( H = 60 \), generates 216,000 (= 24*60*60) ticks per hour

Real world, relative error \( 10^{-5} \), generates 216,000 (= 24*60*60) ticks per hour

Maximum drift rate \( p := 1 - p \); \( dC_p/dt = 1 + p \)

If clocks drift in the opposite direction, max 2p \( \Delta t \) apart

No clocks differ more than \( \delta \): resynchronize (in software) at least every \( \delta/2p \) (must have \( 2p \delta t < \delta \) )
Physical Clocks

- How to synchronize clocks

  **Internal synchronization**: Synchronize them with each other
  For a synchronization bound $D > 0$, $|C_i(t) - C_j(t)| < D$

  **External synchronization**: Synchronize them with real world clocks, say a source $S$ of UTC-time.
  For a synchronization bound $D > 0$, $|S(t) - C_i(t)| < D$

- If a system is externally synchronized with bound $D$ then it is internally synchronized with bound $2D$

---

**Cristian’s Algorithm**

There is a time server (WWV receiver)
Goal: have all other machines synchronized with it (external synchronization)
1. Periodically with period $T < \delta/2\pi$, each machine asks the time server for the current time.
2. The server responds asap with the current time, $C_{UTC}$
3. The client sets its clock to $C_{UTC}$

---

**Problems**

1. Time must never run backwards, why? (Monotonicity condition)
   $t' > t \Rightarrow C_p(t') > C_p(t)$

   Introduce changes gradually
   How,
   Say a clock generates 100 interrupts per sec, an interrupt adds 10msec
   Advance the clock: 11 msec
   Slow down the clock: 9 msec

---

**The Berkeley Algorithm**

(internal synchronization)
1. A time daemon periodically polls every machine to ask the time
2. Each machine replies
3. Based on the answers, computes an average. Informs every machine to advance or slow down its clock
**A Decentralized Algorithm**

Divide time into fixed-length \( R \) resynchronization intervals

\( i \)-th interval: \([T_0 + iR, T_0 + (i+1)R)\), \( T_0 \) some agreed-upon time instance in the past

Each machine:
1. At the beginning of each interval, broadcasts its current time (note, these broadcasts will not happen precisely simultaneously, why?)
2. Starts collecting all other broadcasts that arrive during an interval \( S \)
3. Runs an algorithm (e.g., average; discard \( m \) highest and \( m \) lower values and average the rest) to compute a new time from them

**Use of Synchronized Clocks**

New algorithms that utilize synchronized clocks

Example: Enforcing at-most-once message delivery, even in the face of crashes

Traditional approach: each message bears a unique message number (the server stores all message number it has seen. Problem, if the server crashes and reboots, also how long to keep message numbers)

Modified approach: each message carries a connection identifier (chosen by the server) + a timestamp (its local time)

For each connection (i.e., sending process), the server records the most recent timestamp (that is, the largest timestamp) it has seen

Any incoming message for a connection with a timestamp that is lower than the stored timestamp is rejected as duplicate.

**Logical Clocks**

**Lamport Timestamps**

It suffices that two processes agree on the order in which events occur (no need to synchronize their clocks)

The happens-before relation

\( a \) happens-before \( b \), \( a \rightarrow b \): means that each process agrees that first event \( a \) occurs, then afterwards event \( b \) occurs.

Two cases, where happens-before can be directly observed:

1. If process \( p_i \) send \( a \rightarrow b \), then \( a \rightarrow b \) (that is if \( a \) and \( b \) are events in the same process, and \( a \) occurs before \( b \) then \( a \rightarrow b \) is true)
2. If \( a \) is the event of a message being sent by one process and \( b \) is the event of the message being received by another process, then \( a \rightarrow b \) is true. (For any message \( m \), \( m \) = send(m) \rightarrow receive(m))

Transitive relation. If \( a \rightarrow b \) and \( b \rightarrow c \), then \( a \rightarrow c \).

**Use of Synchronized Clocks**

To determine, when to remove a timestamp, each server maintains a variable \( G \)

\[ G = \text{CurrentTime} - \text{MaxLifeTime} - \text{MaxClockSkew} \]

\( \text{MaxLifeTime} \) (how long a message can live)

\( \text{MaxClockSkew} \) (synchronization bound among clocks)

Write \( G \) to disk every \( \Delta t \)

During recovery, reload \( G \), increment it by \( \Delta t \)

**Lamport Timestamps**

If \( e \) and \( e' \) are events, and \( e \rightarrow e' \), then we can find a series of events \( e_j \)

\( j = 1, 2, ..., n \), either case 1 or case 2 applies between \( e_j \) and \( e_{j+1} \) (that is, either they occur in succession in the same process, or there is a message \( m \) such that \( e_j = \text{send}(m) \) and \( e_{j+1} = \text{receive}(m) \))

\( \ast \) The sequence of events \( e_1, e_2, ..., e_n \) may not be unique.

Example:

\[ a \rightarrow b, e \rightarrow f \]

Case 1: \( a \rightarrow b \rightarrow c, d \rightarrow e, e \rightarrow f \)

Case 2: \( b \rightarrow c, \text{case 1} \) or \( b \rightarrow c, d \rightarrow e, \text{case 2} \)

What about \( e \) and \( a \)?

Two events, \( a \) and \( b \), such that neither \( a \rightarrow b \) nor \( b \rightarrow a \) holds are said to be concurrent (happens-before is a partial order).
Distributed Systems, Spring 2004

Goal: For every event a, assign a time value \( L \) (Lamport timestamp) such that all processes agree on it.

Property of \( L \): If \( a \rightarrow b \), then \( L(a) < L(b) \).

L must always go forward (increasing).

Algorithms for assigning timestamps to events:

Each process \( p_i \) maintains its own logical clock \( L_i \).

A Lamport logical clock is a monotonically increasing counter used to apply Lamport timestamps to events. (we denote them \( L_i(e) \) or \( L(e) \)).

1. \( L_i \) is incremented before each event is issued: \( L_i = L_i + 1 \).
2. (a) When a process sends a message \( m \), it also sends a timestamp \( t = L_i \).
   (b) When a message \((m, t)\) arrives at the receiver process \( p_j \), then \( p_j \) sets \( L_j = \max(L_j, t) \) and before timestamping the event receive\((m)\) applies rule 1.

Example:

It can be shown that:
For any two events \( a \) and \( b \), \( a \rightarrow b \Rightarrow L(a) < L(b) \).

The converse is not true. For instance in the example, \( L(b) > L(e) \) but \( b \) and \( e \) are concurrent.

Example:

Account = 1000, \( p_1 \) adds 100, \( p_2 \) increments by 1%. 
Replica1: 1111  Replica2: 1110

Example: A database replicated across several sites
Issue: update operations must be performed in the same order at each copy, so that all copies are exactly the same.
Example:
Account = 1000, \( p_1 \) adds 100, \( p_2 \) increments by 1%
Replica1: 1111  Replica2: 1110

Totally-Ordered Multicast:

Requirement of a totally-ordered multicast:
A multicast operation by which all messages are delivered in the same order to each receiver.

Assumption: reliable (no message lost) FIFO (messages from the same sender are received in the order they are sent) delivery of messages.

When a message is multicast, it is conceptually also sent to its sender.

Each message is timestamped with the current (logical) time of its sender.

Process \( p_i \) sends timestamped messages \( m_{i,g} \) to all others. (puts message in a local queue queue)

Process \( p_j \) receives \( m_{i,g} \).

A process \( p_j \) can deliver a queued message \( m_{i,g} \) to an application, only when:

1. The message is at the head of the queue.
2. For each process \( p_k \) where there is a message \( m_{i,g} \) in queue, with a larger timestamp (i.e., the message has been acknowledged by each other process).
**Vector Clocks**

Goal: overcome the fact that we cannot conclude the order of events from the values of their timestamps, that is, from \( L(a) < L(b) \), we cannot conclude that \( a \rightarrow b \).

A vector clock for a system of \( N \) processes is an array of \( N \) integers.

Each process keeps its own vector clock \( V_i \), which it uses to timestamp local events. Processes add vector timestamps on the messages they send:

1. Initially, \( V_i[j] = 0 \), for \( i, j = 1, 2, \ldots, n \)
2. Just before \( p_i \) timestamps an event, it sets \( V_i[i] = V_i[i] + 1 \)
3. \( p_i \) includes the value \( t = V_i \) in every message it sends (the whole vector)
4. When \( p_i \) receives a message with timestamp \( t \), it sets \( V_i[j] = \max(V_i[j], t[j]) \) for \( j = 1, 2, \ldots, n \) (that is, it takes the component-wise maximum of two vector timestamps, known as a merge operation)

For a vector clock \( V_i \), \( V_i[i] \) is the number of events that \( p_i \) has timestamped and \( V_i[j] \) for \( i \neq j \) is the number of events that have occurred at \( p_j \) that \( p_i \) has potentially been affected by.

**Example**

How to compare vector timestamps:

\[ V = V' \iff V[j] = V'[j] \quad \text{for } j = 1, 2, \ldots, n \]

\[ V \leq V' \iff V[j] \leq V'[j] \quad \text{for } j = 1, 2, \ldots, n \]

\[ V < V' \iff V[j] \leq V'[j] \text{ and } V \neq V' \]

It can be shown that:

For any two events \( a \) and \( b \), \( a \rightarrow b \iff L(a) < L(b) \)

The converse also holds, \( L(a) < L(b) \implies a \rightarrow b \)

For instance in the example, \( b \) and \( e \) are concurrent which can be also concluded by the fact that neither \( V(e) \leq V(b) \) nor \( V(b) \leq V(e) \)

**Histories and States**

- **History**: a series of events that occur at each process.
- **State**: the state of a process immediately after the \( k \)th event occurred.
- **Initial State**: the state of each process before any events. The initial state is a local state of each process, plus messages currently in transit.
Global State

Global State (or distributed snapshot)

Which states are meaningful, which combination of process states could have occurred at the same time?

Corresponds to initial prefixes of the individual process histories

A cut of the system's execution is a subset of its global state that is a union of prefixes of process histories

$$C = h_0^c \cup h_1^c \cup \ldots \cup h_{N-1}^c$$

Are all cuts acceptable?

Say $$e^1$$ is the sending of a message and $$e^2$$ is the receipt.

The actual execution never was in a global state corresponding to the process states at that frontier, examine the relation about events

A cut $$C$$ is consistent if, for each event it contains, it also contains all the events that happened-before that event,

For all events $$e$$, if $$f \rightarrow e$$, then $$f \in C$$

A consistent global state is one that corresponds to a consistent cut

The execution of a distributed system as a transition between global states of the system

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$$

In each transition, precisely one event occurs at some single process of the system

- A run is a total ordering of all events in a global history that is consistent with each local history's ordering.
- A consistent run or linearization is an ordering of the events in a global history that is consistent with the happened-before relation on $$H$$

Not all runs pass through consistent global states, but all linearizations do

A state $$S'$$ is reachable from state $$S$$ if there is a linearization that passes through $$S$$ and $$S'$$

Testing for properties amounts for evaluating a global state predicate

A global state predicate is a function that maps from the set of global states of processes in the system to $$\{True, False\}$$.

Stable properties: once True at a state, remain True for all future states reachable from that state

Two interesting properties:

- Suppose $$a$$ is an undesirable property (e.g., deadlock)
- Safety with respect to $$a$$ is the assertion that a evaluates to False for all states $$S$$ reachable from $$S_0$$

Conversely, let $$\beta$$ be a desirable property (e.g., reaching termination)

Liveness with respect to $$\beta$$ is the property that, for any $$linearization$$ $$L$$ starting in state $$S_0$$, $$\beta$$ evaluates to True for some state $$S_L$$ reachable from $$S_0$$

The Chandy and Lamport Snapshot Algorithm

Goal: record a set of process and channel states

If a message has been sent by a process $$P$$ but not received by a process $$Q$$, we consider it part of the channel between them

Assumptions:

- Neither channels nor processes fail
- Reliable communication, any message sent is received exactly once
- Unidirectional channels, FIFO-ordered message delivery
- There is a path between any two processes
- The processes may continue their execution and send and receive messages while the snapshot algorithm takes place
Global State

**The Chandy and Lamport Snapshot Algorithm**

Any process, say P, initiates the algorithm:

- P records its own state
- P sends a marker along each of its outgoing channels

**Process Q:**

- When Q receives a marker through incoming channel C
  - If it has not saved its local state,
    - Records it, starts recording all incoming messages
    - Sends a marker along each of its outgoing channels
  - Else,
    - Stops recording the state of channel C (state of C from R to Q: Q records any message on C that arrived after Q recorded its state and before the sender (R) recorded its own state)
- Finishes when it has received and processed a marker along each of its incoming channels

**Example**

Q receives marker for first time

**Example (continued)**

Q records its local state and sends markers along each of its outgoing edges

Q records all incoming messages

Q finishes recording the state of incoming channel

**Note**

Records a consistent state but one that may never have occurred at the same time

**Termination of the snapshot algorithm**

**Proof**

We assume that a process that has received a marker records its state within a finite time and sends markers over each outgoing channel within a finite time.

- If there is a path of communication channels and processes from p_i to p_j, then p_j will record its state a finite time after p_i recorded its state.

Since the graph is strongly connected, it follows that all processes will record their states and the states of their incoming channels a finite time after some process initially records its state.

The algorithm selects a cut from the history of execution

We shall prove that this cut is consistent.

**Proof**

Let e_i and e_j be events occurring at p_i and p_j respectively such that e_i → e_j.

We need to show that if e_i is in the cut then e_j is also in the cut.

For the purposes of contradiction, assume that e_i is not in the cut, that is, p_i recorded its state before e_i occurred.

Let m_1, m_2, ..., m_k the sequence of messages that lead to e_j.

By FIFO ordering, the marker from p_i would have reached p_j before these messages, thus p_j would have recorded its state before event e_j.

This contradicts our assumption that e_i is in the cut.
We shall prove a reachability relation between the observed global state and the initial and final states when the algorithm runs.

Let:
- $S_{\text{init}}$: the global state immediately before the first process recorded its state.
- $S_{\text{snap}}$: the global state when the snapshot algorithm terminates (immediately after the last state recording action).
- $S_{\text{final}}$: the recorded global state.
- $S_{\text{sys}}$: a linearization of the system as it executed (actual execution).

We shall show that there is a permutation of $S_{\text{sys}}$, $S_{\text{sys'}} = e'_0, e'_1, e'_2, \ldots$ such that all three states, $S_{\text{init}}$, $S_{\text{snap}}$, and $S_{\text{final}}$ occur in $S_{\text{sys'}}$.

**Proof.**

Categorize all events in $S_{\text{sys}}$ as pre-snap and post-snap events.

A pre-snap event at process $p_i$ is one that occurred at $p_i$ before $p_i$ recorded its state. All other events are post-snap.

(Note: a post-snap event may occur before a pre-snap event in $S_{\text{sys}}$, if the two events belong to different processes.)

Suppose $e_j$ is a post-snap event at one process and $e_{j+1}$ is a pre-snap event at a different process.

It cannot be that $e_j \rightarrow e_{j+1}$ (why?)

Thus, we can swap the two events without violating the happened-before relation. We continue swapping until all pre-snap events $e'_0, e'_1, e'_2, \ldots, e'_R-1$ are ordered prior to all post-snap events $e'_R, e'_{R+1}, e'_{R+2}, \ldots$

$S_{\text{snap}} = e'_0, e'_1, e'_2, \ldots, e'_R-1$

$S_{\text{init}}$, $S_{\text{final}}$

---

**Example:**

Take a snapshot for detecting termination of a computation.

**How?** Use the snapshot algorithm.

When $Q$ receives the marker for the first time, considers the process that sent that marker as its predecessor.

When $Q$ completes, sends its predecessor a DONE message.

When the initiator of the distributed snapshot receives a DONE from all its successors, the snapshot has been completely taken.

Problem: incoming messages.

We need a snapshot in which all channels are empty.

---

**Topics to be covered**

Clock Synchronization
Logical Clocks
Global State
Election Algorithms
Mutual Exclusion
Distributed Transactions

---

**Election Algorithms**

The Bully Algorithm
A Ring Algorithm
The Bully Election Algorithm

1. P sends an ELECTION message to all processes with higher numbers
2. If no one responds, P wins the election and becomes the coordinator
3. If one of the higher-ups answers, it takes over.

Assumes:
• Reliable message delivery, but processes may crash
• That the system is synchronous (assumes timeouts to detect a process failure)
• Each process knows which processes have higher identifiers and can communicate with them

Example

The Bully Election Algorithm

- Process 6 tells 5 to stop
- Process 6 wins and tells everyone

When 7 comes back, it holds an election

Example (continued)

The Ring Election Algorithm

1. Any site P may initiate the procedure.
2. Each site:
   • Sends an ELECTION message to its successor, adds its number in the list
   • If the successor is down, the sender skips over the successor and goes to the next member along the ring
3. When the message arrives at the initiating site P (how is this detected?) P circulates a COORDINATOR message with the higher number in the list as the coordinator

Assumption: each site knows its successor in the ring

Example

Mutual Exclusion

A Centralized Algorithm
A Distributed Algorithm
A Token-Ring Algorithm
Mutual Exclusion

To read or update shared data structures, enter a critical region (CR) to achieve mutual exclusion.

In centralized systems: semaphores, monitors, etc

Essential requirements for mutual exclusion:

Safety: At most one process may execute in the CR at a time.

Liveness: Requests to enter and exit the CR eventually succeed.

Liveness implies freedom of deadlocks and starvation (indefinite postponement of entry for a process that has requested it).

Absence of starvation is a fairness condition.

Another fairness condition: order in which processes enter the CR.

The order that processes enter the CR follows their requests to enter the CR:

- If one request to enter the CR happened-before another, then entry to the CR is granted in that order.

A Centralized Mutual Exclusion Algorithm

Select one process as the coordinator:

- To enter a CR, send a <request> message to the coordinator.
- If no other process in the CR, the coordinator sends a <grant> message.
- Else, denies permission (e.g., does not reply and thus blocks the requesting process, or send a deny message).
- Upon exiting a CR, send a <release> message to the coordinator. The coordinator grants access to another process (e.g., takes the first item of the queue and sends a grant message).

Correct (safety): Guarantees mutual exclusion?
Fair: No starvation? Order?
Easy to implement

But the coordinator is a single point of failure & a performance bottleneck/no way to distinguish a dead coordinator from "permission denied"

Ricart and Agrawala’s algorithm

- Requires that there be a total order of all events in the system.
  (This can be achieved by using for example the Lamport’s algorithm for providing timestamps)
- Assumes reliable sending of messages (i.e., every message is acknowledged)
A Decentralized Mutual Exclusion Algorithm

When a process wants to enter the CR,
- builds a `<request>` message \( M = (\text{CR-id}, \text{process-number}, \text{timestamp}) \)
- sends the message to all other processes (including itself)

Upon receipt of a `<request>` message \( M \)
1. If the receiver is not in the CR and does not want to enter the CR, replies `<OK>`
2. If the receiver is in the CR, it does not reply, queues \( M \)
3. Else (the receiver is not in the CR, but wants to enter the CR), compares the timestamp with the timestamp of its own request, if lower, replies `<OK>`, else does not reply, queues \( M \)

Waits till it receives OK from all processes

Upon exit from a CR,
- sends OK to all processes in its queue
- deletes them from the queue

Correct: guarantees mutual exclusion
No deadlock or starvation

However, worst than the centralized solution:
- Number of messages: \( 2(n-1) \)
- \( N \) points of failure! If a process fails, all others are blocked

Solution?
- Each process must maintain a list with all other processes
- Load balancing?

Slight improvement: Enter the CR, when granted permission from the majority (to work, a process after granting permission to a process, cannot grant permission to another one)

A Token-Ring Mutual Exclusion Algorithm

When the ring is initialized, process 0 is given a token.
The token circulates the ring

When a process \( k \) acquires the token:
- If it wants to enter the CR, it enters the CR, does all the work, leaves the region, passes the ring to \( k+1 \)
- Else, it just passes the ring to \( k+1 \)

Correctness (safety)?
Starvation?

Problems:
Last token
Process crashes: require acknowledging the receipt of a token
### Distributed Transactions

**Comparison**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Client delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed</td>
<td>2 ((n-1))</td>
<td>2 ((n-1))</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to (\infty)</td>
<td>0 to (n-1)</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

Messages per entry/exit determine the bandwidth consumed.

System throughput (the rate at which the collection of processes as a whole can access the critical region).

It is based on the synchronization delay between one process exiting the critical region and the next process entering it (not shown in the Table above).

---

**The Transaction Model**

#### Classification of Transactions

- Nested transaction
- Distributed transaction

#### Implementation

- Concurrency Control

---

**Distributed Transactions**

The Transaction Model

Examples of primitives for transactions:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

**The ACID properties**

- **A** - Atomicity
- **C** - Consistency
- **I** - Isolation
- **D** - Durability

---

**The Transaction Model**

### Examples of primitives for transactions.

- BEGIN_TRANSACTION
- END_TRANSACTION
- ABORT_TRANSACTION
- READ
- WRITE

---

**The Transaction Model**

#### Classification of Transactions

- Nested transaction
- Distributed transaction

---

**The Transaction Model**

### Examples of primitives for transactions.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

**The ACID properties**

- **A** - Atomicity
- **C** - Consistency
- **I** - Isolation
- **D** - Durability

---

**The Transaction Model**

#### Classification of Transactions

- Nested transaction
- Distributed transaction

---

**The Transaction Model**

### Examples of primitives for transactions.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

**The ACID properties**

- **A** - Atomicity
- **C** - Consistency
- **I** - Isolation
- **D** - Durability

---

**The Transaction Model**

### Examples of primitives for transactions.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

**The ACID properties**

- **A** - Atomicity
- **C** - Consistency
- **I** - Isolation
- **D** - Durability
a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing

Log
\[ x = 0 / 1 \]
\[ y = 0 / 2 \]
\[ x = 1 / 4 \]
\[ y = y + 2 \]
\[ x = y * y \]

FORTRAN
x = 0;
y = 0;
BEGIN_TRANSACTION;
x = x + 1;
y = y + 2;
x = y * y;
END_TRANSACTION;

(a) (b) (c) (d)
Strict two-phase locking.

Concurrency control using timestamps.