Synchronization (contd.)


Hongfei Yan
School of EECS, Peking University
3/17/2008

Outline
- Mutual Exclusion
  - Permission-based
  - Token-based
- Election Algorithms
  - The Bully Algorithm
  - A Ring Algorithm
  - Superpeer Selection
- Distributed Transactions

Terminology
- In concurrent programming a critical section is a piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread of execution.
- Mutual exclusion (ME, often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

Applications use ME
- Replicated databases
  - Protocols that control access to replicated data and ensure data consistency in case of network partitioning are called replica control protocols.
  - A transaction (an access request to data) is the basic unit of user computation in a database.
  - All replica control protocols require that mutual exclusion must be guaranteed between two write operations and a read and write operation.
- Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory.

Performance Metrics of DME Algorithm
- Message Complexity (MC)
  - The number of messages exchanged by a process per CS entry.
- Synchronization Delay (SD)
  - the average time delay in granting CS, which is the period of time between the instant a site invokes mutual exclusion and the instant when it enters CS.
- Fault tolerance, Availability, Safe, Live, Fair.

Mutual Exclusion
- Prevent simultaneous access to a resource
- Two basic kinds:
  - Permission based
    - A Centralized Algorithm
    - A Decentralized Algorithm
  - Token based
    - A Token Ring Algorithm
**Centralized Algorithm**
- Use a central coordinator to simulate how it is done in a one-processor system

**A Centralized Algorithm**
1. Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.
2. Process 2 then asks permission to access the same resource. The coordinator does not reply.
3. When process 1 releases the resource, it tells the coordinator, which then replies to 2.

**Distributed Hash Table**
- Nodes and names have keys, which are large integers. You get the key from the name by hashing. Nodes get keys (called IDs) by some way, usually just random.
- Given a name, hash it to the key.
- Now find the node that is responsible for that key. It is the node that has an ID >= key.

**Structured Peer-to-Peer Architectures**
- The mapping of data items onto nodes in Chord.

**Definition of variables for node n, using m-bit identifiers**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>finger[k].start</td>
<td>((n+2^k-1) \mod 2^m, 10 \leq k \leq m)</td>
</tr>
<tr>
<td>.interval</td>
<td>([\text{finger}[k].\text{start}, \text{finger}[k+1].\text{start}])</td>
</tr>
<tr>
<td>.node</td>
<td>first node (\leq n.\text{finger}[k].\text{start})</td>
</tr>
<tr>
<td>successor</td>
<td>the next node on the identifier circle; (\text{finger}[1].\text{node})</td>
</tr>
<tr>
<td>predecessor</td>
<td>the previous node on the identifier circle</td>
</tr>
</tbody>
</table>

**Distributed Hash Tables**
- Resolving key 26 from node 1 and key 12 from node 28 in a Chord system.
A Decentralized Algorithm

- Use a distributed hash table (DHT).
  - Hashes to a node.
- Each resource has $n$ coordinators (called replicas in the book). A limit $m (> n/2)$ is pre-defined.
- A client acquires the lock by sending a request to each coordinator.
  - If it gets $n$ permissions, then it gets it.
  - If a resource is already locked, then it will be rejected (as opposed to just blocking.)

1. Send lock requests
3. Release if failed.

Coordinator Failure

- If a coordinator fails, replace it.
- But what about the lock state?
- This amounts to a resetting of the coordinator state, which could result in violating mutual exclusion.
- How many would have to fail?
  - $2m - n$
- What is the probability of violation?

Probability of Violation

- Let $p$ be the probability of failure during some time $\Delta t$.
- The probability that $k$ out of $n$ coordinators reset is:
  \[
  P[k] = \binom{n}{k} p^k (1 - p)^{n-k}
  \]
- To violate mutual exclusion, you need at least $2m - n$ failures.
  \[
  \sum_{k=2m-n}^{n} P[k]
  \]
- With node participation for 3 hours, $\Delta t$ of 10 seconds, and $n = 32$ and $m = 0.75n$, the probability of violation is less than $10^{-40}$.

A Distributed Algorithm

1. When a process wants a resource, it creates a message with the name of the resource, its process number, and the current (logical) time.
2. It then reliably sends the message to all processes and waits for an OK from everyone.
3. When a process receives a message:
   a. If the receiver is not accessing the resource and is not currently trying to access it, it sends back an OK message to the sender.
   - "Yes, you can have it. I don't want it, so what do I care?"
   b. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
   - "Sorry, I am using it. I will save your request, and give you an OK when I am done with it."
   c. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.
     - "I want it also, but you were first."
     - "Sorry, I want it also, and I was first."
   d. When done using a resource, send an OK on its queue and delete them all from the queue.

Two processes (0 and 2) want to access a shared resource at the same moment.
- Process 0 has the lowest timestamp, so it wins.
- When process 0 is done, it sends an OK also, so 2 can now go ahead.
Evaluation
- How many messages are required? More or less than centralized?
  - One request and OK from everyone else, so 2(n-1).
- More scalable? How much work per node, per lock?
- Is it better than centralized? How many points of failure?
  - We have replaced a poor one with a worse one.
- Can we figure out how to handle failure?

A Token Ring Algorithm
- (a) An unordered group of processes on a network.
- (b) A logical ring constructed in software.

A Comparison of the Four Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>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>Decentralized</td>
<td>3rel, k = 1, 2,...</td>
<td>2 m</td>
<td>Starvation, low efficiency</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 m</td>
<td>0 to n - 1</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

Election Algorithms
- Principle: An algorithm requires that some process acts as a coordinator. The question is how to select his special process dynamically.
- Note: In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions => single point of failure.
- Question: If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
- Question: Is a full distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?
### Election by Bullying

**Principle:** Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator.

**Issue:** How do we find the heaviest process?

Any process can just start an election by sending an election message to all other processes (assuming you don’t know the weights of the others).

If a process $P_{\text{heavy}}$ receives an election message from a lighter process $P_{\text{light}}$, it sends a take-over message to $P_{\text{light}}$. $P_{\text{light}}$ is out of the race.

If a process doesn’t get a take-over message back, it wins, and sends a victory message to all other processes.

---

### The Bully Algorithm (1)

![Diagram of the Bully Algorithm (1)](image)

**Election in a Ring**

**Principle:** Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.

- When it gets back to the initiator, everyone had a chance to make its presence known.

The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

**Question:** Does it matter if two processes initiate an election?

---

### The Bully Algorithm (2)

![Diagram of the Bully Algorithm (2)](image)

**Question:** We’re assuming something very important here—what?
Superpeer Selection

- Issue: How do we select superpeers such that:
  - Normal nodes have low-latency access to superpeers
  - Superpeers are evenly distributed across the overlay network
  - There is a predefined fraction of superpeers
  - Each superpeer should not need to serve more than a fixed number of normal nodes
- **DHT**: Reserve a fixed part of the ID space for superpeers.
- **Example**: if \( S \) superpeers are needed for a system that uses \( m \)-bit identifiers, simply reserve the \( k = \lceil \log_2 S \rceil \) leftmost bits for superpeers. With \( N \) nodes, we’ll have, on average, \( 2^k - mN \) superpeers.
- **Routing to superpeer**: Send message for key \( p \) to node responsible for \( p \ AND \ 11 \cdots 100 \cdots 00 \)

Elections in Large-Scale Systems

- Moving tokens in a two-dimensional space using repulsion forces.

Outline

- Election Algorithms
- Mutual Exclusion
- Distributed Transactions
  - Transaction model
  - Classification of transactions
  - Implementation
  - Concurrency control

Transactions

- The most important reliability technology for client-server systems
- Now start an in-depth examination of the topic
  - How transactional systems really work
  - Implementation considerations
  - Limitations and performance challenges
  - Scalability of transactional systems

Transactions on a single database:

- In a client/server architecture,
- A transaction is an execution of a single program of the application (client) at the server.
  - Seen at the server as a series of reads and writes.
- We want this setup to work when
  - There are multiple simultaneous client transactions running at the server.
  - Client/Server could fail at any time.

Client-Server Computing

- 99% of all distributed systems use client-server architectures!
- Discuss stateless and stateful architectures
- Review major database system issues
Client-Server Concept
- Server program is shared by many clients
- RPC protocol typically used to issue requests
- Server may
  - manage special data,
  - run on an especially fast platform,
  - or have an especially large disk
- Client systems
  - handle "front-end" processing
  - and interaction with the human user

Server and Its Clients

Examples of Servers
- Network file server
- Database server
- Network information server
- Domain name service
- Kerberos authentication server

Summary of Typical Split
- Server deals with
  - bulk data storage,
  - high perf. computation,
  - collecting huge amounts of background data that may be useful to any of several clients
- Client deals with
  - the "attractive" display,
  - quick interaction times
  - Use of caching to speed response time

Statefulness Issues
- Client-server system is stateless if:
  - Client is independently responsible for its actions,
  - server doesn’t track set of clients or ensure that cached data stays up to date
- Client-server system is stateful if:
  - Server tracks its clients, takes actions to keep their cached states "current",
  - Client can trust its cached data.

Best Known Examples?
- The UNIX NFS file system is stateless.
  Bill Joy: "Once they replaced my file server during the evening while my machine was idle. The next day I resumed work right where I had left off, and didn’t even notice the change!"
- Database systems are usually stateful.
  Client reads database of available seats on plane, information stays valid during transaction
### Typical Issues in Design

- Client is generally simpler than server:
  - may be single-threaded,
  - can wait for reply to RPC's
- Server is generally multithreaded,
  - designed to achieve extremely high concurrency and throughput.
  - Much harder to develop
- Reliability issue:
  - if server goes down, all its clients may be "stuck".
  - Usually addressed with some form of backup or replication.

### Use of Caching

- In stateless architectures, cache is responsibility of the client.
  - Client decides to remember results of queries and reuse them.
  - Example: caching Web proxies, the NFS client-side cache.
- In stateful architectures, cache is owned by server.
  - Server uses "callbacks" to its clients to inform them if cached data changes, becomes invalid.
  - Cache is "shared state" between them.

### Example of stateful approach

- Transactional software structure:
  - Data manager holds database
  - Transaction manager does begin op1 op2 ... opn commit
  - Transaction can also abort; abort is default on failure
- Transaction on database system:
  - Atomic: all or nothing effects
  - Consistent: the transaction does not violate system invariants
  - Isolated: concurrent transactions do not interfere with each other
    - i.e., Concurrent transaction execution should be equivalent (in effect) to a serialized execution.
  - Durable: once committed, results are persistent

### Comments on Transactions

- Well matched to database applications
- Requires special programming style
- Typically, splits operations into **read** and **update** categories.
  - Transactional architecture can distinguish these
  - Idea is to run transactions concurrently but to make it look as if they ran one by one in some sequential (serial) order

### Why are Transactions Stateful?

- Client knows what updates it has done, what locks it holds. Database knows this too
- Client and database **share** the guarantees of the model. See consistent states
- Approach is free of the inconsistencies and potential errors observed in NFS

### The Transaction Model

- Updating a master tape is fault tolerant
Transactional Execution Model

- Applications are coded in a stylized way:
  - `begin` transaction
  - Perform a series of `read`, `update` operations
  - Terminate by `commit` or `abort`
- Example 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>

E.g., Reserving a Seat on Plane

- Example primitives for transactions

<table>
<thead>
<tr>
<th>BEGIN_TRANSACTION</th>
<th>reserve WP -&gt; JFK; reserve JFK -&gt; Nairobi; reserve Nairobi -&gt; Malindi; END_TRANSACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>reserve WP -&gt; JFK; reserve JFK -&gt; Nairobi; reserve Nairobi -&gt; Malindi; ABORT_TRANSACTION</td>
</tr>
</tbody>
</table>

- Transaction to reserve three flights commits
- Transaction aborts when third flight is unavailable

Classification of Transactions

- Flat Transactions
- Nested Transactions
- Distributed Transactions

Distributed Transactions

- A nested transaction
- A distributed transaction

Implementation of Transactions

- Private Workspace
  - When a process starts a transaction, it is given a private workspace containing all the files to which it has access.
  - Until the transaction either commits or aborts, all of its reads and writes go to the private workspace.
- Writeahead Log
  - Files are actually modified in place, but before any block is changed, a record is written to a log telling which transaction is making the change, which file and block is being changed, and what the old and new values are.
**Writeahead Log**

- **x = 0;**
- **y = 0;**
- **BEGIN_TRANSACTION:**
  - **x = x + 1;**
  - **y = y + 2**
  - **x = y * y;**
- **END_TRANSACTION:**

(a) A transaction
(b) – (d) The log before each statement is executed

**Concurrency Control of Transactions**

- Serializability
- Two-Phase locking
- Pessimistic timestamp ordering

**Concurrency Control (1)**

- General organization of managers for handling transactions.

**Concurrency Control (2)**

- General organization of managers for handling distributed transactions.

**Need for serializable execution**

- General organization of managers for handling transactions.

**Non serializable execution**

- Problem: transactions may “interfere”. Here, T2 changes x, hence T1 should have either run first (read and write) or after (reading the changed value).
Serializable execution

Data manager interleaves operations to improve concurrency but schedules them so that it looks as if one transaction ran at a time. This schedule “looks” like $T_2$ ran first.

Serializability

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
<th>Schedule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 0$; $x = x + 1$; $x = 0$; $x = x + 2$; $x = 0$; $x = x + 3$</td>
<td>$x = 0$; $x = x + 1$; $x = x + 2$; $x = 0$; $x = x + 3$</td>
<td>Illegal</td>
</tr>
</tbody>
</table>

Transaction must have a lock on each data item it will access.
- Gets a “write lock” if it will (ever) update the item
- Use “read lock” if it will (only) read the item. Can’t change its mind!
- Obtains all the locks it needs while it runs and hold onto them even if no longer needed
- Releases locks only after making commit/abort decision and only after updates are persistent

Strict Two-phase locking (1)

Suppose that $T'$ will perform an operation that conflicts with an operation that $T$ has done:
- $T'$ will update data item $X$ that $T$ read or updated
- $T$ updated item $Y$ and $T'$ will read or update it
- $T$ must have had a lock on $X/Y$ that conflicts with the lock that $T'$ wants
- $T$ won’t release it until it commits or aborts
- So $T'$ will wait until $T$ commits or aborts
Two-phase locking is “pessimistic”
- Acts to prevent non-serializable schedules from arising: pessimistically assumes conflicts are fairly likely
- Can deadlock, e.g. T1 reads x then writes y; T2 reads y then writes x. This doesn’t always deadlock but it is capable of deadlocking
  - Overcome by aborting if we wait for too long,
  - Or by designing transactions to obtain locks in a known and agreed upon ordering

Contrast: Timestamped approach
- Using a fine-grained clock, assign a “time” to each transaction, uniquely. E.g. T1 is at time 1, T2 is at time 2
- Now data manager tracks temporal history of each data item, responds to requests as if they had occurred at time given by timestamp
- At commit stage, make sure that commit is consistent with serializability and, if not, abort

Example of when we abort
- T1 runs, updates x, setting to 3
- T2 runs concurrently but has a larger timestamp. It reads x=3
- T1 eventually aborts
- ... T2 must abort too, since it read a value of x that is no longer a committed value
  - Called a cascaded abort since abort of T1 triggers abort of T2

Pros and cons of approaches
- Locking scheme works best when conflicts between transactions are common and transactions are short-running
- Timestamped scheme works best when conflicts are rare and transactions are relatively long-running
- Weihl has suggested hybrid approaches but these are not common in real systems

Summary
- Mutual Exclusion
  - Permission-based
  - Token-based
- Election Algorithms
  - The Bully Algorithm
  - A Ring Algorithm
  - Superpeer Selection
- Distributed Transactions

References
- Chapter 5 of [Tanenbaum, 2002]
- Chapter 5 & 24 of [Birman, 2005]
  - book1, [Coulouri, 2005]
  - book2, [Birman, 2005]
  - book3, [Tanenbaum, 2006]
  - book4, [Tanenbaum, 2002]
  - ......