Concurrency and Synchronization

Today
- Doing the right thing at the right time
- Mutual exclusion
- Election algorithms
- Transactions
Coordinating processes

- How do you coordinate processes actions or make them agree on a value?
  - Distributed mutual exclusion
  - Leader election
  - Concurrency control

- Assumptions
  - Reliable channels – messages are “eventually” delivered
  - Processes are independent of each other
  - Process fail only by crashing
    - How do you know? Ask a failure detector
    - What if the system is asynchronous? Failure detector can only “suspect” failure
Mutual exclusion

- Processes want exclusive access to some resource
  - There are not shared variables nor a single kernel
  - All communication is through message passing
  - Application-level protocol
    
    ```
    enter() // enter critical section
    resourceAccess()
    exit() // exit critical section
    ```

- Essential requirements
  - ME1 (safety): At most 1 process in the CS
  - ME2 (liveness): Request to enter/exit the CS are eventually granted

- Fairness requirement sometimes made
  - ME3 (→ ordering): Requests to enter the CS are granted in happened-before order
Mutual exclusion algorithm and metrics

- **Common algorithms**
  - Central server
  - Ring based
  - Multicast and logical clocks (Ricart & Agrawala ‘81)
  - Maekawa’s voting algorithm

- **Comparing**
  - Bandwidth consumed – proportional to number of messages sent in entry/exit
  - Client delay – at each entry/exit operation
  - Fault tolerance
Centralized

- A central server grants permission
  - To enter – process sends request to server and waits for OK
  - Server upon request – if nobody is in CS, let them go
    else, hold reply and queue the process
  - Server upon release – choose oldest request and send OK to
    process

<table>
<thead>
<tr>
<th>ME1 – Mutual exclusion</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME2 – No starvation</td>
<td>✔</td>
</tr>
<tr>
<td>ME3 - Ordering</td>
<td>✗</td>
</tr>
</tbody>
</table>
Ring-based algorithm (token-based)

- Organize processes in a *logical ring*
- Exclusion is granted by having a token
- Token is passed around the ring
  - Don’t need it anymore? Pass it to the next

<table>
<thead>
<tr>
<th>ME1 – Mutual exclusion</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME2 – No starvation</td>
<td>✓</td>
</tr>
<tr>
<td>ME3 - Ordering</td>
<td>✗</td>
</tr>
</tbody>
</table>
Ricart and Agrawala

- **To get entry**
  - Multicast request
  - Wait until getting N-1 replies
  - Now you have it

- **Upon getting a request**
  - If somebody has it or another process’ request is first, queue request, else reply immediately

- **At exit**
  - Reply to queued requests

<table>
<thead>
<tr>
<th>ME1 – Mutual exclusion</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME2 – No starvation</td>
<td>✔</td>
</tr>
<tr>
<td>ME3 - Ordering</td>
<td>✔</td>
</tr>
</tbody>
</table>
Maekawa’s voting

- Candidate process: must collect sufficient votes to enter to the critical section
- Each process $p_i$ maintain a voting set $V_i$ ($i=1, \ldots, N$), where $V_i \subseteq \{p_1, \ldots, p_N\}$
- Sets $V_i$: chosen such that $\forall \; i,j$
  - $p_i \in V_i$
  - $V_i \cap V_j \neq \emptyset$ (at least one common member of any two voting sets)
  - $|V_i| = k$ (fairness, all voting sets of the same size)
  - Each process $p_j$ is contained in $M$ of the voting sets $V_i$
- Makeawa showed that an optimal solution
  - $K \sim \sqrt{N}$ and $M = K$
Maekawa’s voting

- For $p_i$ to enter the CS
  - Multicast request to all processes in $V_i$
  - Wait until you get $K$ replies and you have it

- On receipt of a request from $p_i$ at $p_j$
  - If it has it or voted already
    - Queue request without replying
  - Else send reply and set voted is true

- For $p_i$ to exit the CS
  - Set state to released and multicast this to all processes in $V_i$

- On receipt of a release from $p_i$ at $p_j$
  - If queue is non-empty, reply to one (*happened-before to avoid deadlocks*) and set voted to true
  - Else set voted to false

<table>
<thead>
<tr>
<th>ME1 – Mutual exclusion</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME2 – No starvation</td>
<td>✓</td>
</tr>
<tr>
<td>ME3 - Ordering</td>
<td>✓</td>
</tr>
</tbody>
</table>
# Algorithm comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Msgs. per entry/exit</th>
<th>Delay before entry (msgs)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Ring-based</td>
<td>1 to infinite</td>
<td>0 to N-1</td>
<td>Lost token, process crash</td>
</tr>
<tr>
<td>Ricart/Agrawala</td>
<td>2(N-1)</td>
<td>2(N-1)</td>
<td>Process crash</td>
</tr>
<tr>
<td>Maekawa’s</td>
<td>$3\sqrt{N}$</td>
<td>$2\sqrt{N}$</td>
<td>Crash of a voting process</td>
</tr>
</tbody>
</table>
Election algorithms

- Many distributed algorithms require one process to act as coordinator
- In general, it doesn’t matter which one – so pick the one with the largest ID/weight
- Elections conclude when all agree on new coordinator
- Each process $p_i$ maintains the identity of the elected in the variable $Elected_i$ (or NIL)
- Properties to satisfy: $\forall p_i$,
  - E1 Safety: $Elected_i = \text{NIL}$ or $Elected = P$
  - E2 Liveness: $p_i$ participates and sets $Elected_i \neq \text{NIL}$, or crashes
A ring algorithm

- Somebody, \( p_i \), notice coordinator is down and calls an election.
- \( p_i \) sends ELECTION message with its number in to first successor up.
- Recipient forward messages adding itself as candidate.
- Who started it all, will eventually receive a message with itself in the list; elect coordinator and inform all (ELECTED message).
- ELECTED messages goes around the ring once.
A ring algorithm

- E1 is met – a process has to receive its own message back before sending ELECTED around so all processes before must have lower numbers.
- E2 follows from the guaranteed traversal of the ring.
The Bully algorithm

- Somebody, $p_i$, notice coordinator is down and calls an election
- $p_i$ sends ELECTION message to all processes with higher numbers
- If no-one responds, $p_i$ is the winner
  - Algorithm assumes synchronous system – so you can build a reliable failure detector
- If a process with a higher number receives the ELECTION message, reply with OK and calls an election
- When done, winner let everybody know with an ELECTED message

Garcia-Molina, ‘82
The Bully algorithm

- E1, assuming no process is replaces, is satisfied; not to processes will think they are the coordinator since the one with lower number will defer to the leader
- E2 works by the assumption of reliable message delivery

If 7 ever wakes up, it will call for elections
## Algorithm comparison

<table>
<thead>
<tr>
<th>Election algorithm</th>
<th>Num. of messages</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual ring</td>
<td>3N-1</td>
<td>Doesn’t tolerate faults</td>
</tr>
<tr>
<td>Bully</td>
<td>N-2 to O(N²)</td>
<td>System must be synchronous</td>
</tr>
</tbody>
</table>
Transaction model

- Closely related to mutual exclusion – transactions
  - Protect shared resource – data – against simultaneous access by concurrent processes
- A bit more – allow processes to access and modify multiple objects as a single atomic operation – atomic
- Originally from business – *once we signed the contract, you got to deliver*
- Classical example – a client moving $ bet/ accounts
  1. Withdraw an amount a from account 1
  2. Deposit that amount a in account 2

System or client connection goes down right here
Transaction model

- Using transactions requires special primitives
  - Supplied by either the language or the underlying distributed systems
  - `BEGIN_TRANSACTION` and `END_TRANSACTION` – to delimit the scope of a transaction; `END_TRANSACTION` terminates and tries to commit
  - `ABORT_TRANSACTION` – Kill it and restore the old value
  - `READ` and `WRITE`

- All-or-nothing is just one of the ACID properties
  - Atomic – to the outside world, they happen indivisibly
  - Consistent – does not violate system invariants (if the invariant hold before, it holds after)
  - Isolated or serializable – Concurrent transactions don’t interfere with each other
  - Durable – once committed, the changes are permanent
Transactions and nested transactions

- So far, flat transactions – some limitations
  - Do not allow partial results to be committed or aborted; its strengths is partly its weakness
  - Imagine trying to reserve a multi-hop plane trip as a transaction

- Nested transactions
  - Transactions made of sub-transactions
  - But what if a sub-transaction commits and the parent aborts?
  - Semantics are clear – when any transaction/sub-transaction starts it is conceptually given a copy of of the data fro the entire system
    - If it aborts, private universe vanishes
    - If it commits, private replaces the parent’s universe
  - So, if the parent abort, all sub-transactions abort too
  - Great for distributed systems – logical division
Nested and distributed transactions

- A logical division of a nested transaction does not mean distribution is taken care of.
- Nested transaction – a transaction logically decomposed into a hierarchy of sub-transactions.
- Distributed transaction – logically flat but operates on distributed data.

**Diagram:**
- Nested transaction: Decomposed into sub-transactions, each operating on a separate database.
- Distributed transaction: Operates on a single distributed database, showing two physically separated parts of the same database.
Implementing transactions

- Clearly modifying the only copy will make hard to ensure atomicity
- Two common alternatives
  - Private workspace
    - Get your own copy at start – clearly not optimal as such
    - Don’t need a copy for reads
    - Save making copies until you modify
    - Keep shadow copies until commit
  - Writeahead log
    - Modify copies in place but keep track of what you do in a log
    - If commit, write commit to log
    - If abort, use the log to rollback
Concurrent control

• Goal of concurrency control – to allow several transactions to execute simultaneously

• Best understood in terms of three different managers
  – Data manager – performs the actual read/write on data; knows nothing about transactions
  – Scheduler – controls concurrency, it determines which transaction is allowed at what time to pass a read/write to the data manager; doing this ensures isolation and consistency
  – Transaction manager – guarantees atomicity; process transactions by translating them into scheduling request for the scheduler
Serializability

- Goal of concurrency control – allow concurrent execution while ensuring isolation
- Three transactions
  - Outcome, depending on the order in which they are run, could be $x$ being 1, 2 or 3

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

- Possible schedules
  Sched 1: $x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3; (x = 3)$
    ✔
  Sched 2: $x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3; (x = 3)$
    ✔
  Sched 3: $x = 0; x = 0; x = x + 1; ; x = 0; x = x + 2; x = x + 3; (x = 5!)$
    ✗
Serializability

- The actual computation is not important, only matters that, in the example, the value of x is being changed
  - We can represent transactions as a series of read/writes

```
BEGIN_TRANSACTION
x = 0;
end = x + 1;
END_TRANSACTION
```

Concurrency control – schedule conflicting operations
- Two operations conflict if they operate on the same data item and at least one is a write
  - Read-write or write-write are conflict; read-read are not
  - It doesn’t matter if the conflicting operations are from the same or different transactions

Approaches to concurrency
- Through a mutual exclusion mechanism or explicit ordering
- Optimistic or pessimistic
Locking

- Oldest and most widely used
- Need to read/write an item – get a lock
- Scheduler needs to grant and release locks to ensure serializable schedules

One example – two-phase locking
  - Get all the locks needed in the growing phase
  - Release them in the shrinking phase
  - Once the scheduler has released a lock on behalf of transaction T, it will not grant another lock to T
  - If the transaction release all locks until commit – strict 2PL
    - All lock acquisitions and releases can be handled by the system w/o the transaction being aware of them
    - Eliminates cascade aborts

- Clearly, deadlock is an issue
Timestamp ordering

- Assign each transaction a timestamp at start
  - Each operation carries the timestamp
  - Every data time has a read and a write timestamp

- Supposed scheduler receives an operation read(T,x) from transaction T
  - If ts < ts_{wr}(x) – i.e., a write on x was done after T started – abort T
  - If ts > ts_{wr}(x) – let ts_{rd}(x) = max{ts, ts_{wr}(x)}

- Supposed scheduler receives an operation write(T,x) from transaction T
  - If ts < ts_{rd}(x) – i.e., x was read by a more recent transaction – abort T
  - If ts > ts_{rd}(x) – let ts_{wr}(x) = max{ts, ts_{wr}(x)}

- When a transaction encounters a larger timestamps, it aborts – does not wait as in 2PL → no deadlocks

- Optimistic variation – do whatever, worry later
Summary

- Synchronization is about doing the right thing at the right time …
- What’s the right time?
  - An issue when you don’t share clocks
- What’s the right thing to do?
  - Who can access what when?
  - Who is in charge?