Mutual exclusion

- Suppose nodes occasionally need access to a shared resource, but accessing it simultaneously creates problems
  - E.g. printing a document, updating a record in shared memory

- We need a protocol that allows nodes to request access, and that guarantees that such nodes eventually get exclusive access.
  - Nodes call enter() to request access
  - Nodes call exit() to release the resource

```c
while true
  enter(); /* critical section */
  exit(); /* non critical section */
```

Advanced Network Security
3. Distributed Algorithms:
Mutual Exclusion

Jaap-Henk Hoepman

Digital Security (DS)
Radboud University Nijmegen, the Netherlands

xotoxot // jhh@cs.ru.nl // www.cs.ru.nl/~jhh

Victorian swimmingpool
Mutual exclusion

- Mutual exclusion: there is at most one node in the critical section
- Progress: if there is at least one node enters, and the critical section is empty, then one of these nodes will eventually get access to the critical section
- No starvation: if a node enters, and if all nodes that get access to the critical section release it, then it will eventually get access

Assumption:
- A fair scheduler,
- Atomic shared variables

Mutual exclusion using message passing (1)

- Using logical clocks [Lamport]
  - Every message carries a timestamp \( t_i \), equal to the logical clock value \( L_i(e) \) of the send event \( e \); recall this induces a total order \( \leq \).
  - Every node maintains a request queue of \( (t, r) \) pairs, initially \( q_i \) (i.e. node \( i \) initially holds the resource) ordered by \( r \) as well.
- (1) To request a resource, node \( i \) sends a request message to all other nodes, including itself.
- (2) When a node \( j \) receives a request message with timestamp \( t_i \) from node \( i \):
  - Node \( j \) adds \( (t_i, r) \) to its request queue.
  - Node \( j \) sends a (timestamped) acknowledgement message back to \( i \).

Mutual exclusion using message passing (2)

- (3) To release a resource, node \( i \) sends a release message to all other nodes, including itself.
- (4) When a node \( j \) receives a release message with timestamp \( t_i \) from node \( i \) it removes any \( (t', r) \) messages from its request queue.
- (5) A node \( i \) is granted access to the resource if
  - (i) Node \( i \)'s request queue contains \( (t', r) \) ordered using \( \leq \) before any other elements \( (t'', r) \) in the request queue.
  - (ii) Node \( i \) has received a message from all other nodes with a timestamp \( t'' > t' \)
Why does this work?

- **Mutual exclusion**
  - Rule 1 and 2 ensure that resource requests are added to all queue's
  - Rule 3 guarantees that a node must have learned about earlier requests before honouring its own
  - Requests are only removed from the queue when the corresponding node releases it (and sends release messages to all nodes) due to rule 3 and 4
- **Progress:** In fact, requests are honoured in the order in which they are made
  - Follows from the fact that requests are ordered by w and served in that order because of rule 5.i and the above
- **No starvation:** Every request is eventually honoured
  - For every request, a node receives an acknowledgement (rule 2), thus rule 5.ii is eventually satisfied
  - Every node eventually releases the resource, then rule 3 and 4 guarantee that eventually any older timestamps are removed from the request queue

Mutual exclusion with shared memory: the easy solution

- **Use test-and-set / read-modify-write**
- **Cf lamport's logical clock based algorithm**

Mutual exclusion: first try

- **Two shared variables**
  - Flags written by 0 and read by 1
  - Flags written by 1 and read by 0
- **Protocol**

```
<table>
<thead>
<tr>
<th>Flag</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Entry</td>
</tr>
<tr>
<td>s</td>
<td>Exit</td>
</tr>
</tbody>
</table>
```
- **What can go wrong?**
- **What if we first set the flags, before testing their value?**
Mutual exclusion: Lamport’s bakery algorithm

- Each node \( i \) maintains two shared variables that it writes and that all other nodes can read
  - \( \text{num}_i \): unbounded
  - \( \text{choosing}_i \)
- Idea: take numbered ticket like in the bakery
  - Except that you have to ask everyone in the shop what their number is, and take the maximum + 1
  - And you should wait for people that haven’t picked a number yet
- Lowest number is next allowed in the critical section

```java
num[] = 0
choosing[] = false
while true
  do while choosing[] = true
      num[] = 1 + max(num[])
      choosing[] = false
  for i ≠ j
      do while choosing[j] do /* wait */
      while (num[j] ≥ 0) ∧ (num[j]) < (num[]) do /* wait */

/* critical section */
num[] = 0
```

Proof of bakery algorithm: mutual exclusion

- Lemma 1: If in C.S. and there is a i.e. \( \text{num}_i \) ≠ 0 then \( \text{num}_j \) > \( \text{num}_i \)
  - Before entering / first waited until choosing[j] = false and then waited until either (num[i] = 0) or (num[j]) > (num[i])
  - If num[i] = 0, then by assumption that now num[i] ≠ 0, node \( i \) must have changed num[i] after node \( j \) read it (i.e., read num[j] ≠ write num[j]). Node \( i \) set its current value of num[i] before reading choosing[j] = false and reading num[j] (i.e., write num[j] = read choosing[j] = false → read num[j]). But node \( i \) sets choosing[j] = true before reading num[j] and writing num[j] (i.e., write choosing[j] = true → read num[j] = write num[j]).
  - This can only happen if read choosing[j] = false → write choosing[j] = true. Hence write num[j] = read num[j] no node \( k \) must have seen this value for num[j] when computing a ticket. By the protocol \( k \) sets num[k] to a larger value
  - If (num[j]) > (num[i]) then either node \( i \) did not change the value, or it entered again and by the same argument as above sets num[i] to a larger value
Proof of bakery algorithm: mutual exclusion

Lemma 1: if i is in C.S. and there is a j s.t. $\text{num}(i) = 0$ then $\text{num}(i), j > 0$

Lemma 2: for all i, $\text{num}(i) \geq 0$
   - Follows from the protocol

Lemma 3: if i is in C.S. then $\text{num}(i) > 0$
   - Follows from lemma 2 and the fact that i chooses $\text{num}(i) = 1 + \text{max}_{j \neq i} \text{num}(j)$

Theorem: the bakery protocol satisfies mutual exclusion
   - Suppose not. Then for i \neq j we have $\text{num}(i) > 0$ and $\text{num}(j) > 0$ by lemma 3 and then both $(\text{num}(i), i) > (\text{num}(j), j)$ and $(\text{num}(j), j) > (\text{num}(i), i)$ by lemma 1. A contradiction.

Proof of bakery algorithm: progress

Suppose C.S. is empty and there are nodes entering
   - Note: the exit() protocol will eventually be completed
   - Every node that enters gets a ticket
   - Let i be the node with minimal $(\text{num}(i), i)$
   - It will never see nodes with smaller tickets
   - It only needs to wait for nodes that choose, but these always complete after some number of steps (non blocking operation)
   - It eventually enters the C.S., so we have progress.

Proof of bakery algorithm: no starvation

In general we have that if progress holds, and if any node i is waiting at most a finite number of other nodes can enter the C.S., then no starvation holds

Lemma 4: Let i have a ticket. Any node entering after that will have a bigger ticket. Hence at most $n - 1$ nodes can have a smaller ticket and beat node i.

Because progress holds and lemma 4 holds, no starvation then also holds.