Goals of the lecture

• Time domain vs Causality domain

• Lamport’s Mutual Exclusion Algorithm

• Formal Verification
  • Key Lemmas
  • Safety
  • Liveness
  • Fairness

References: Lamport 79,  
Garg and Tomlinson 94
Time domain vs Causality domain

- most problems require causality domain
  - accounts for variable execution schedule
- problems in causality domain easier
  - mutual exclusion
  - ordering of messages
  - observing a global property
Properties of the Mutual Exclusion Algorithm

- a fixed number of processes
- a shared resource called the critical section (CS).
- Task is to coordinate processes.
- Requirements are:
  Safety: Two processes should not use the CS simultaneously.
  Liveness: Every request for the CS is eventually granted.
  Fairness: Requests must be granted in the order they are made.
Formal Specification

Lamport’s algorithm assumes that all channels are FIFO

\[ s < t \land s \rightarrow u \land t \rightarrow v \Rightarrow \neg(v < u) \]

- \( \text{req}(s) = P_{s,p} \) has requested the critical section
- \( \text{cs}(s) = P_{s,p} \) has permission to enter the critical section in \( s \)
- Cooperation assumption:
  \[ \text{cs}(s) \Rightarrow (\exists t : s < t : \neg \text{req}(t)) \]
Formal Requirements

\[ s \| t \Rightarrow \neg (cs(s) \land cs(t)) \]  
(Safety)

\[ req(s) \Rightarrow (\exists t :: s < t \land cs(t)) \]  
(Liveness)

\[ \text{req}\_\text{start}(s) = \text{min}\{t \mid s < t \land cs(t)\} \]

\[ \text{req}\_\text{start}(s) = \text{req}(s) \land \neg \text{req}(s.\text{prev}) \]

\[ \text{req}\_\text{start}(s) = P_{s.p} \text{ made a request for the CS in state } s. \]

\[ (\text{req}\_\text{start}(s) \land \text{req}\_\text{start}(t) \land s \rightarrow t) \Rightarrow \text{next}\_\text{cs}(s) \rightarrow \text{next}\_\text{cs}(t) \]  
(Fairness)

- \text{next}\_\text{cs}(s) \text{ and } \text{next}\_\text{cs}(t) \text{ exist due to liveness.}
- \text{next}\_\text{cs}(s) \text{ and } \text{next}\_\text{cs}(t) \text{ are not concurrent due to safety.}
Informal Specification of the Mutual Exclusion Algorithm

- **request CS**: send a timestamped message to all other processes and add a timestamped request to the queue.

- **On receiving a request**: the request and its timestamp is stored in the queue and an acknowledgment is returned.

- **To release the CS**: send a release message to all other processes.

- **On receiving a “release”**: delete the corresponding request from the queue.

\[
P_3 \circ \overline{\text{req}(21, 1), \cdots}
\]

\[
P_1 \circ \overline{\text{req}(21, 1), \text{ack}(24, 2), \text{ack}(25, 3), \cdots}
\]

\[
P_2 \circ \overline{\text{req}(21, 1), \cdots}
\]
Informal Specification [Contd.]

- can access CS if
  - it has a request in the queue with timestamp $t$, and
  - $t$ is less than all other requests in the queue, and
  - it has received a message from every other process with timestamp greater than $t$.

$$P_3 \circ [\text{req}(21,1), \text{req}(24,2) \cdots]$$

$$P_1 \circ \text{req}(21,1), \text{req}(24,2), \text{ack}(25,3) \cdots$$

$$P_2 \circ \text{req}(21,1), \text{req}(24,2) \cdots$$
Formal Description

- Local variables in each state $s$:
  
  $s.q[1..n] : \text{integer initially } \infty$

  $s.v : \text{DDClock}$

- To request the critical section in $t$ where $s \prec_1 t$:
  
  $t.q[t.p] = s.v[t.p]$

  for all $j : j \neq t.p : \text{send “request” to } P_j$

- On receiving “request” in state $t$ sent from state $u$ ($u \sim t$):
  
  $t.q[u.p] = u.q[u.p]$

  send ack to $u.p$

- To release the critical section in state $t$:
  
  $t.q[t.p] = \infty$

  for all $j \neq t.p, \text{send “release” to } P_j$

- On receiving “release” sent from state $u$:
  
  $t.q[u.p] = \infty$
Formal Description [Contd.]

State $s$ has permission to access the critical section when

- there is a request from $P_{s.p}$ with timestamp less than all other requests
- and $P_{s.p}$ has received a message from every other process with a timestamp greater than the timestamp of its own request.

Formal description of $CS(s) \equiv$

$$\forall j : j \neq s.p : (s.q[s.p], s.p) < (s.v[j], j) \land (s.q[s.p], s.p) < (s.q[j], j).$$
Proof of Correctness

We define the predicate

\[ msg(s,t) \equiv (\exists u, t' : u \sim t' \land u < s \land t < t') \]

That is, there exists a message which was sent by \( P_{s.p} \) before \( s \) and received by \( P_{t.p} \) after \( t \).

**Lemma 1** Assume FIFO. \( \forall s, t : s.p \neq t.p : s \not\rightarrow t \land \neg msg(s,t) \Rightarrow t.q[s.p] = s.q[s.p] \).

The following Lemma is crucial in proving the safety property.

**Lemma 2** \( \forall s, t : s.p \neq t.p : s \not\rightarrow t \land s.q[s.p] < t.v[s.p] \Rightarrow t.q[s.p] = s.q[s.p] \)
Safety Property

Lemma 3 (Safety) $s.p \neq t.p \land s||t \Rightarrow \neg(cs(s) \land cs(t))$.

Proof: We will show that $(s||t) \land cs(s) \land cs(t)$ implies false.

Case 1: $t.v[s.p] < s.q[s.p] \land s.v[t.p] < t.q[t.p]$

We get the following cycle.

$\langle s.q[s.p] < \{ cs(s) \land s.p \neq t.p \} s.v[t.p] < \{ this \ case \} t.q[t.p] < \{ cs(t) \land s.p \neq t.p \} t.v[s.p] < \{ this \ case \} s.q[s.p] \rangle$. 

©Vijay K. Garg

Distributed Systems Spring 96
Safety Property [Contd.]

Case 2: \( s.q[s.p] < t.v[s.p] \land t.q[t.p] < s.v[t.p] \)
We get the following cycle.

\[
\begin{align*}
& s.q[s.p] \\
< & \{ cs(s) \land s.p \neq t.p \} \\
& s.q[t.p] \\
= & \{ t.q[t.p] < s.v[t.p], t \not\leftrightarrow s, \text{Lemma 2} \} \\
& s.v[t.p] \cdot s.q[t.p] \cdot t.v[s.p] \cdot t.q[s.p] \\
& \cdot t.q[t.p] \\
< & \{ cs(t) \land s.p \neq t.p \} \\
& t.q[s.p] \\
= & \{ s.q[s.p] < t.v[s.p], s \not\leftrightarrow t, \text{Lemma 2} \} \\
& s.q[s.p].
\end{align*}
\]
Safety Property [Contd.]

Case 3: \( s.q[s.p] < t.v[s.p] \land s.v[t.p] < t.q[t.p] \)

We get the following cycle.

\[
\begin{align*}
s.q[s.p] \\
< \{ \text{cs}(s) \land s.p \neq t.p \} \\
s.v[t.p] \\
< \{ \text{this case} \} \\
t.q[t.p] \\
< \{ \text{cs}(t) \land s.p \neq t.p \} \\
t.q[s.p] \\
= \{ s.q[s.p] < t.v[s.p], s \not\rightarrow t, \text{Lemma 2} \} \\
s.q[s.p].
\end{align*}
\]

Case 4: Similar to case 3.
Liveness Property

Lemma 4 (Liveness) $req(s) \Rightarrow \exists t : s < t \land cs(t)$

Proof: $req(s)$ is equivalent to $s.q[s.p] \neq \infty$. $s.q[s.p] \neq \infty$ implies that there exists $s_1 \in P_{s.p}$ such that $s_1.v[s.p] = s.q[s.p] \land event(s_1) = request$. We show existence of the required $t$ with the following two claims:

Claim 1:
$\exists t_1 : \forall j \neq s.p : t_1.v[j] > s.q[s.p] \land s.q[s.p] = t_1.q[s.p]$

Claim 2:
$\exists t_2 : \forall j \neq s.p : t_2.q[j] > s.q[s.p] \land s.q[s.p] = t_2.q[s.p]$
Fairness Property

Lemma 5 (Fairness) \((\text{req\_start}(s) \land \text{req\_start}(t) \land s \rightarrow t) \Rightarrow (\text{next\_cs}(s) \rightarrow \text{next\_cs}(t))\)

Proof:
Let \(s' = \text{next\_cs}(s)\) be state in which critical section is acquired, and let \(s''\) be state which it is released. Let \(t' = \text{next\_cs}(t)\).
Let \(r\) be the state in \(P_{t.p}\) which received the request message sent from \(s\).
**Fairness Property [Contd.]**

We know the following facts:

1. \( r \leq t \), due to FIFO channels.
2. \( t.v[t.p] = t.q[t.p] \), due to request event at \( t \).
3. \( s.v[s.p] < t.v[t.p] \), since \( s \rightarrow t \) (DD2).
4. \( s.q[s.p] = s.v[s.p] \), due to request event at \( s \).
5. \( r.q[s.p] = s.q[s.p] \), due to receiving request at \( r \).
6. \( r.q[s.p] < t.q[t.p] \), from 2, 3, 4, 5.
7. \( t.q[t.p] = t'.q[t.p] \), by defn of \( t' \).
8. \( t'.q[t.p] \leq t'.q[s.p] \), since \( cs(t') \).
9. \( r.q[s.p] < t'.q[t.p] \leq t'.q[s.p] \), from 6, 7, 8.

This means that \( q[s.p] \) must be increased between \( r \) and \( t' \). That can only happen when \( P_{t.p} \) receives the release message sent from \( s'' \). Thus \( s'' \rightarrow t' \). And since \( s' \rightarrow s'' \), we conclude \( s' \rightarrow t' \).