Goals:
• Introduce the concept of mutual exclusion (in distributed setting)
• Talk about how to share a variable between distributed processes

Reading:
• P. Sivilotti, *Introduction to Distributed Algorithms*, Chapter 7
Summary: Time, Clocks, Synchronization

Channel model: FIFO, lossless, directed

Events, system timelines and logical time
- Can’t assume process clocks agree
- Make use of “logical time”
  \[ A \rightarrow B \Rightarrow \text{time}.A < \text{time}.B \]

Vector clocks: \[ A \rightarrow B \equiv \text{vtime}.A < \text{vtime}.B \]
- Keep track of time in each process
- Order relation allows us to know one event occurred before another

Gossip: distribute info to all nodes
- Key problem is understanding when the algorithm has terminated (all nodes idle, no information in channels)
- Use tree structure to track propagation

Diffusing computation properties:
- Safety: invariant \((\text{claim} \Rightarrow \text{term. cond.})\)
- Progress: term. cond. \(\Rightarrow \text{claim}\)

Today: mutual exclusion
The Mutual Exclusion Problem

Control access to a “critical section” (CS)
- Use in situations where no more than one agent can make use of a resource at a time
- Easy to implement in centralized setting
  - E.g. standard mutex libraries in Unix
- Not so easy when there is no central node and no central clock

Example: intersections for self-driving cars
- Safety: no two cars should be in the intersection at the same time
- Progress: all cars should eventually be allowed to go through the intersection

Traditional (human) protocol for mutual exclusion at intersections (4 way stop)
- First person to reach the intersection gets to go first
- If someone is already at the intersection when you arrive, they were first
- If two or more people arrive at the same time, right hand rule applies

Q1: what happens if four people arrive at the same time?

Q2: if [some] cars are self-driving, who decides who reaches intersection “first”?
  - Should self-driving car give way to aggressive human? Even if they break protocol?
Mutual Exclusion Formal Problem Statement

Specification

- Safety: no two users ($U_i$) are in critical section (CS) at the same time
- Progress: strong and weak
  - Weak: some agent will eventually be allowed to enter CS
  - Strong: all agents will get a chance (as long as they keep requesting)

User process protocol

User process ($U_i$) properties

- $NC$ next $NC \lor TRY$
- stable.$TRY$
- $CS$ next $CS \lor NC$
- transient.$CS$

Composition properties:

- TRY next TRY $\lor$ CS
- TRY $\sim$ CS

Mutual Exclusion Layer

property of the user process but not of the composition of user processes & mutex layers
Approaches to Mutual Exclusion

Centralized control process
- Easiest: everyone makes requests to central “allocator”
- Use standard mutex at that point (eg, simple queue)
- Cons: ............................................................

Token ring  Fri
- Use an indivisible token to grant access
- Pass token around in an “efficient” way
- Pros: relatively easy to implement and verify
- Cons: ............................................................

Distributed computation  Today
- Create protocol by which everyone agrees on who is next
- Pros: works for arbitrary topologies
- Cons: slightly more complex to verify (but only need to do once)

Metrics for choosing an approach  Fri
- Response time
- Number of messages required
Related Problem: Distributed Atomic Variables

General question: how can we “synchronize” a variable in a distributed system?

Proposed algorithm:
- Local variables for each agent (i)
  - $x =$ local copy of shared variable
  - $t_i =$ logical clock for agent $i$
  - queue of modify requests
  - list of “known times” for all other processes (why: ___________)
- Agent executes modification request when
  - request has minimum logical time
  - all known times are later than the request time

Key properties that make this work
- All agents agree on request order
- All agents know who has full information

Mutual exclusion is an example of this
- Use synchronized variable to agree on who gets to access critical section
DGC Example: Changing Gear

Verify that we can’t drive while shifting or drive in the wrong gear

- Five components: follower Control, gcdrive Arbiter, gcdrive Control, actuators and network
- Construct temporal logic models for each component (including network)

Asynchronous operation

- Notation: Message\textsubscript{mod,dir} - message to/from a module; Len = length of message queue
- Verify: follower has the right knowledge of the gear that we are currently in, or it commands a full brake.
  - \( \Box (\text{Len(TransResp}_{f,r}) = \text{Len(Trans}_{f,s}) \wedge \text{TransResp}_{f,r}[\text{Len(TransResp}_{f,r})] = \text{COMPLETED} \Rightarrow \text{Trans}_{f} = \text{Trans}) \)
  - \( \Box (\text{Trans}_{f} = \text{Trans} \lor \text{Acc}_{f,s} = -1) \)
- Verify: at infinitely many instants, follower has the right knowledge of the gear that we are currently in, or we have hardware failure.
  - \( \Box \Diamond (\text{Trans}_{f} = \text{Trans} = \text{Trans}_{f,s}[\text{Len(Trans}_{f,s})] \lor \text{HW failure}) \)
Application Example: Trusted Wingman

Problem description
- UAV (unmanned aerial vehicle) flies close as long as high bandwidth link is available
- Assume low speed link is always available

Temporal logic specification
- \( \text{mode} = \text{lost} \rightarrow \text{stable}(d(x_l, x_f) > d_{sep}) \)
  - “Lost mode leads to the distance between the aircraft always being larger than d_{sep}”
  - Need to make sure both aircraft agree that high speed link is lost

Implementation using shared variables
- Implement using distributed variable to keep track of system “mode”
- Also allows extension to multiple aircraft (eg, rest of the formation)
Lamport’s Mutual Exclusion Algorithm

Idea: treat request queue as a distributed atomic variable
• reqQ: queue of timestamps requests for CS (sorted in increasing order)
• knownT: list of last “known times” for other processes
• Actions
  - Request entry: add to reqQ; broadcast <reqi, ti> to all other processes
  - Receive req: add to reqQ; send <acki, ti>
  - Receive ack: update knownT[j]
  - Receive release: remove Uj’s request from reqQ

UNITY program: list of actions that can be executed by each agent (in any order)
• SendReq: mode = NC → mode = TRY || (∀j :: send(i, j, ⟨reqi, ti⟩))
• RecvReq: (∃j :: recv(i, j) = ⟨reqj, tj⟩ → recQ.push/sort(⟨reqj, tj⟩) || send(i, j, ⟨acki, ti⟩))
• RecvAck: (∃j :: recv(i, j) = ⟨ackj, tj⟩ → knownT[j] := tj)
• EnterCS: mode = TRY ^ recQ[head] = ⟨reqi, ti⟩ ^ (∀j :: knownT[j] > ti) → mode = CS;
• ReleaseCS: mode = CS → mode = NC || reqQ.pop(⟨reqi, ti⟩) || (∀j :: send(i, j, ⟨reli, ti⟩))
• RecvRel: (∃j :: recv(i, j) = ⟨reli, tj⟩ → reqQ.pop(⟨reli, tj⟩)

Conditions to enter CS
• L1: req at head of reqQ
• L2: knownT[j] > ti for all other j

To release CS
• remove req from reqQ
• broadcast <releasei> message
Sample Execution

- **{SendReq:} mode = NC → mode = TRY || (\(\forall j :: \text{send}(i, j, \langle \text{req}_i, t_i \rangle)\))**
- **{RecvReq:} \(\exists j :: \text{recv}(i, j) = \langle \text{req}_j, t_j \rangle \rightarrow \text{recQ}.\text{push/sort}(\langle \text{req}_j, t_j \rangle) \ || \ \text{send}(i, j, \langle \text{ack}_i, t_i \rangle)\))**
- **{RecvAck:} \(\exists j :: \text{recv}(i, j) = \langle \text{ack}_j, t_j \rangle \rightarrow \text{knownT}[j] := t_j\))**
- **{EnterCS:} mode = TRY ^ \text{recQ}[\text{head}] = \langle \text{req}_i, t_i \rangle ^ \ (\forall j :: \text{knownT}[j] > t_i) \rightarrow \text{mode} = \text{CS};**
- **{ReleaseCS:} mode = \text{CS} \rightarrow \text{mode} = \text{NC} || \text{reqQ}.\text{pop}(\langle \text{rel}_i, t_i \rangle) \ || \ (\forall j :: \text{send}(i, j, \langle \text{rel}_i, t_i \rangle))**
- **{RecvRel:} \(\exists j :: \text{recv}(i, j) = \langle \text{rel}_j, t_j \rangle \rightarrow \text{reqQ}.\text{pop}(\langle \text{ack}_j, t_j \rangle)\))**

```
{SendReq:} mode = NC \rightarrow mode = TRY || (\(\forall j :: \text{send}(i, j, \langle \text{req}_i, t_i \rangle)\))
{RecvReq:} \(\exists j :: \text{recv}(i, j) = \langle \text{req}_j, t_j \rangle \rightarrow \text{recQ}.\text{push/sort}(\langle \text{req}_j, t_j \rangle) \ || \ \text{send}(i, j, \langle \text{ack}_i, t_i \rangle)\))
{RecvAck:} \(\exists j :: \text{recv}(i, j) = \langle \text{ack}_j, t_j \rangle \rightarrow \text{knownT}[j] := t_j\))
{EnterCS:} mode = TRY ^ \text{recQ}[\text{head}] = \langle \text{req}_i, t_i \rangle ^ \ (\forall j :: \text{knownT}[j] > t_i) \rightarrow \text{mode} = \text{CS};
{ReleaseCS:} mode = \text{CS} \rightarrow \text{mode} = \text{NC} || \text{reqQ}.\text{pop}(\langle \text{rel}_i, t_i \rangle) \ || \ (\forall j :: \text{send}(i, j, \langle \text{rel}_i, t_i \rangle))
{RecvRel:} \(\exists j :: \text{recv}(i, j) = \langle \text{rel}_j, t_j \rangle \rightarrow \text{reqQ}.\text{pop}(\langle \text{ack}_j, t_j \rangle)\))
```
Proof of Correctness

Safety: need to show that no two processes are in CS at the same time

- Assume the converse: Ui and Uj are both in CS
- Both Ui and Uj must have their own requests at head of queue
- Head of Ui: <reqi, ti>
- Head of Uj: <reqj, tj>
- Assume WLOG ti < tj (if not, switch the argument)
- Since Uj is in its CS, then we must have tj < Uj.knownT[i]  
  \[\Rightarrow\] <reqi, ti> must be in Uj.reqQ (since messages are FIFO)
- ti < tj  \[\Rightarrow\] reqj can’t be at the head of Uj.reqQ
- \[\rightarrow\] (contradiction)

Progress: need to show that eventually every request is eventually processed

- Approach: find a metric that is guaranteed to decrease (or increase)
- One metric: number of entries in Ui.knownT that are less than its request time (ti)
  - Represents number of agents who might not have received our request
- Is this a good metric? Check conditions that are needed for induction:
  - Bounded below by zero and if at zero then we eventually enter our critical section
  - Must always decrease as other processes enter their critical section (and someone will execute their CS at some point in time)
Summary: Mutual Exclusion

Key ideas:
- Distributed protocol for allowing access to a shared resource ("critical section")
- Can treat as special case of distributed atomic variables
- **User** process specifications:
  - $NC \xrightarrow{\text{next}} NC \vee TRY$
  - $\text{stable.}TRY$
  - $CS \xrightarrow{\text{next}} CS \vee NC$
  - $\text{transient.}CS$
- **System specifications**:
  - Safety: no two users ($Ui$) are in critical section (CS) at the same time
  - Progress: all agents will get a chance (as long as they keep requesting): $TRY \sim CS$

Good example of *composition* between user and system processes and specs

Friday: optimizations + token-based algorithms