Goals:
• Briefly review where we are at in the course
• 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
Where We Are In the Course

Weeks 1-3: UNITY programs
- Predicate calculus, equivalence, quantification [HW #1]
- Program execution (UNITY semantics) [HW #2]
- Stability properties (next, stable, invariant, unless) [HW #2]
- Progress properties (transient, ensures, leadsto) [HW #3]
- Induction (metrics) and proofs of correctness [HW #3, 4]

Week 4: Intro multi-agent systems
- Logical clocks and vector clocks [HW #4]
- Diffusing computations [HW #5]

Week 5: Snapshots
- Consistent cuts [HW #5]

\[ V = \sum_i (x_i - A)^2 \]
Where We Going In the Course

Week 6: mutual exclusion (for distributed systems)
• How do we insure that a distributed set of agents can access a common resource without deadlock

Week 7: synchronization (for distributed systems)
• How do we synchronize a set of agents to perform a coordinated function
• Example: “dining philosophers”

Week 8: consensus with faults
• How do we expand concepts so far when there might be malicious (or failing) agents present
• Example: “Byzantine generals problem”

Week 9/10 (Thanksgiving): Paxos and distributed databases
• Maintaining consistent distributed databases (including possibility of faulty or malicious agents)
• Example: blockchain
• Note: lectures/homework schedule will be perturbed due to Thanksgiving holiday; details to be posted
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

Composition properties:

\[
\text{TRY} \quad \text{next} \quad \text{TRY} \lor \text{CS} \\
\text{TRY} \sim \text{CS}
\]

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  Wed

- 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  Wed

- 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.
  - $\Diamond ((\text{Len(TransResp}_{f,r}) = \text{Len(Trans}_{f,s}))$
    $\land \text{TransResp}_{f,r}[\text{Len(TransResp}_{f,r})] = \text{COMPLETED} \Rightarrow \text{Trans}_f = \text{Trans})$
  - $\Diamond (\text{Trans}_f = \text{Trans} \lor \text{Acc}_{f,s} = -1)$
- Verify: at infinitely many instants, follower has the right knowledge of the gear that we are currently in, or we have hardware failure.
  - $\Diamond\Diamond (\text{Trans}_f = \text{Trans} =$
    $\text{Trans}_{f,s}[\text{Len(Trans}_{f,s})] \lor \text{HW failure}$)
Lamport’s Mutual Exclusion Algorithm

Idea: treat request queue as a distributed atomic variable

- reqQ: queue of timestamps requests for CS (sorted in ________ 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 = 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) = ⟨relj, tj⟩ → reqQ.pop(⟨relj, tj⟩))
Sample Execution

- \{\text{SendReq:}\} \text{ mode} = \text{TRY} \ || (\forall j :: \text{send}(i, j, \langle req_i, t_i \rangle))
- \{\text{RecvReq:}\} (\forall j :: \text{recv}(i, j) = \langle req_j, t_j \rangle \rightarrow \text{recQ.push/sort}(\langle req_j, t_j \rangle) \ || \text{send}(i, j, \langle \text{ack}_i, t_i \rangle))
- \{\text{RecvAck:}\} (\forall j :: \text{recv}(i, j) = \langle \text{ack}_j, t_j \rangle \rightarrow \text{knownT}[j] := t_j)
- \{\text{EnterCS:}\} \text{ mode} = \text{TRY} \land \text{recQ}[\text{head}] = \langle req_i, t_i \rangle \land (\forall j :: \text{knownT}[j] > t_i) \rightarrow \text{mode} = \text{CS};
- \{\text{ReleaseCS:}\} \text{ mode} = \text{CS} \rightarrow \text{mode} = \text{NC} \ || \text{reqQ.pop}(\langle \text{rel}_q, t_i \rangle) \ || (\forall j :: \text{send}(i, j, \langle \text{rel}_i, t_i \rangle))
- \{\text{RecvRel:}\} (\forall j :: \text{recv}(i, j) = \langle \text{rel}_j, t_j \rangle \rightarrow \text{reqQ.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: \langle reqi, ti \rangle
• Head of Uj: \langle reqj, tj \rangle
• Assume WLOG ti < tj (if not, switch the argument)
• Since Uj is in its CS, then we must have tj < Uj.knownT[i] \implies \langle reqi, ti \rangle must be in Uj.reqQ (since messages are FIFO)
• ti < tj \implies reqj can’t be at the head of Uj.reqQ
• \implies (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:
  - \( NC \text{ next } NC \lor TRY \)
  - stable.\( TRY \)
  - \( CS \text{ next } CS \lor NC \)
  - transient.\( CS \)
- System specifications:
  - Safety: no two users (\( U_i \)) are in the 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

Wednesday: optimizations + token-based algorithms