

THE UNIVERSITY *of* EDINBURGH

Distributed Systems Fall 2026

Yuvraj Patel

Disclaimer: Slides prepared using multiple sources (UW-Madison – Remzi, Andrea, Mike; Cambridge – Martin Kleepman; Distributed Systems book by MVS/ AST); University of Edinburgh – Yuvraj Patel

Scan to check-in



Expires at 06/11/2026 18:10

Today's Agenda

Raft

Mutual Exclusion

Deadlocks

Raft – Consensus Protocol

Designed to be easy to understand

Equivalent to Paxos in fault-tolerance and performance

Decomposed into relatively independent sub-problems

Raft is Paxos optimized for log appends

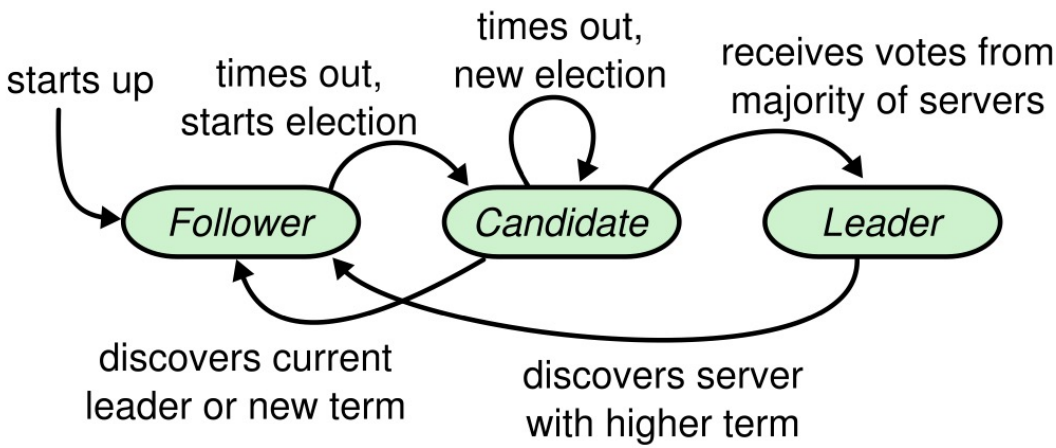
Raft vs Paxos

- Paxos – agrees separately on each client operation
- Raft – agrees on each new leader (and on tail of the log); agreement not required for most client operations

Roles in Raft

A node can be either

- Follower – Passive nodes; They issue no requests on their own; Respond to requests from leaders and candidates
- Candidate – Used to elect a new leader; Transitions from a Follower and transitions to a leader or follower
- Leader – Handles all client requests



Raft – High-Level Understanding

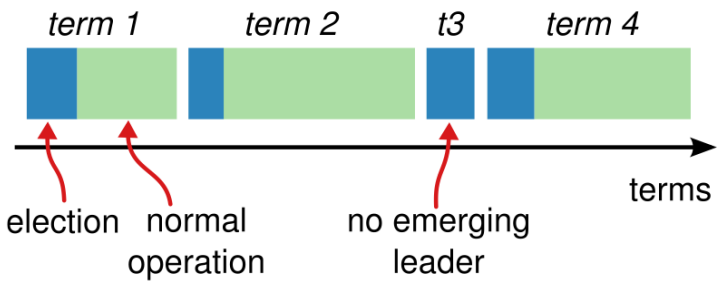
Leader Election

Raft divides time into terms of arbitrary length; terms are numbered consecutive integers

Each term begins with an election, where one or more candidates attempts to become a leader

Two possible outcomes of an election

- Candidates wins with majority; Elected leader for the term
- No election – Split Votes / Split Brain



Leader Election – Normal Scenario

Leader Election – Split Votes

Leader Election

Term acts as a logical clock and helps detect obsolete information such as stale leaders

Each node stores a current term number, increases monotonically

Current terms exchanged while normal communication

- One node's current term smaller than others, it updates it term to larger value
- If leader/candidate discovers its term is out of date; revert to follower role

If node receives a request with a stale term number, reject the request

Log Replication

Log entries over time

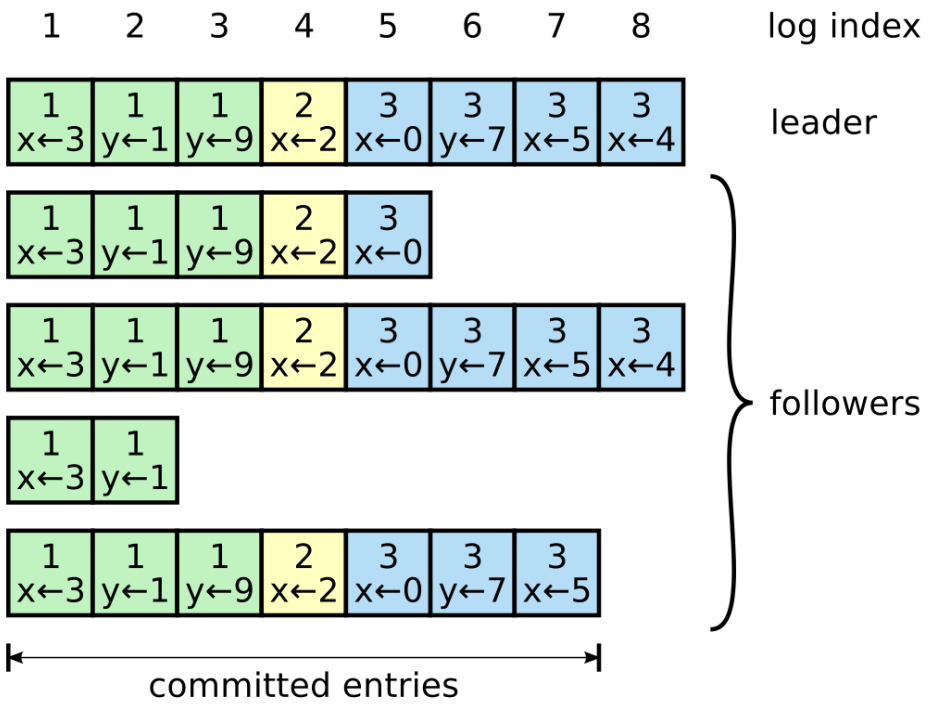
A leader's log is the ultimate truth

While election, ensure that the leader has all committed entries

Leader keeps track of each follower's log

Leader ensures all followers are up to date

- Either remove uncommitted log entries or append to log entries



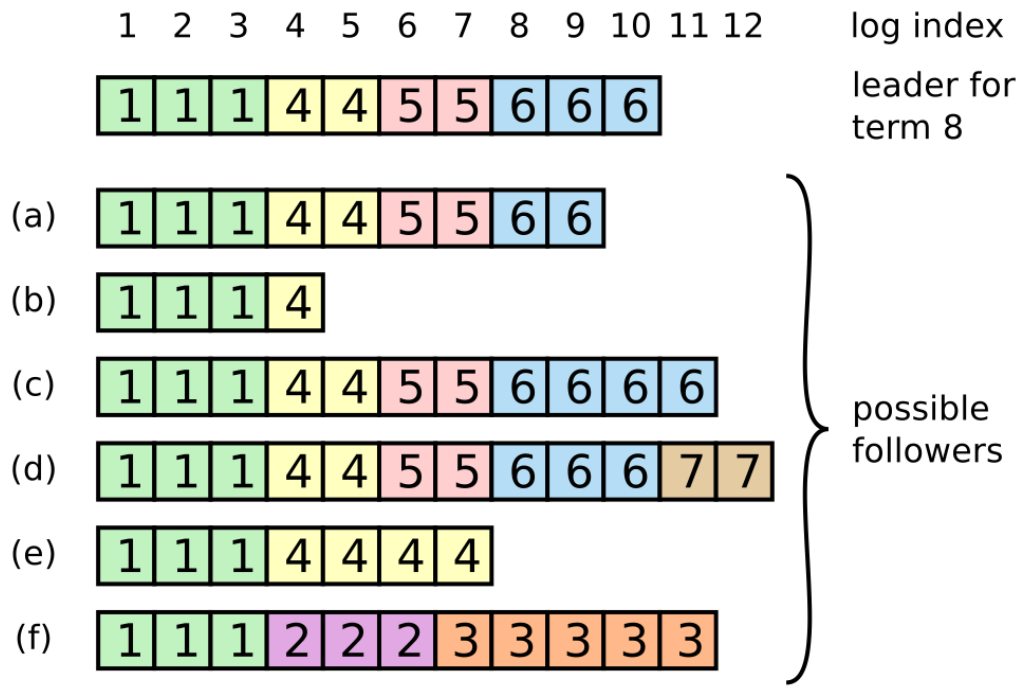
Log entries over time (contd.)

Different possibilities

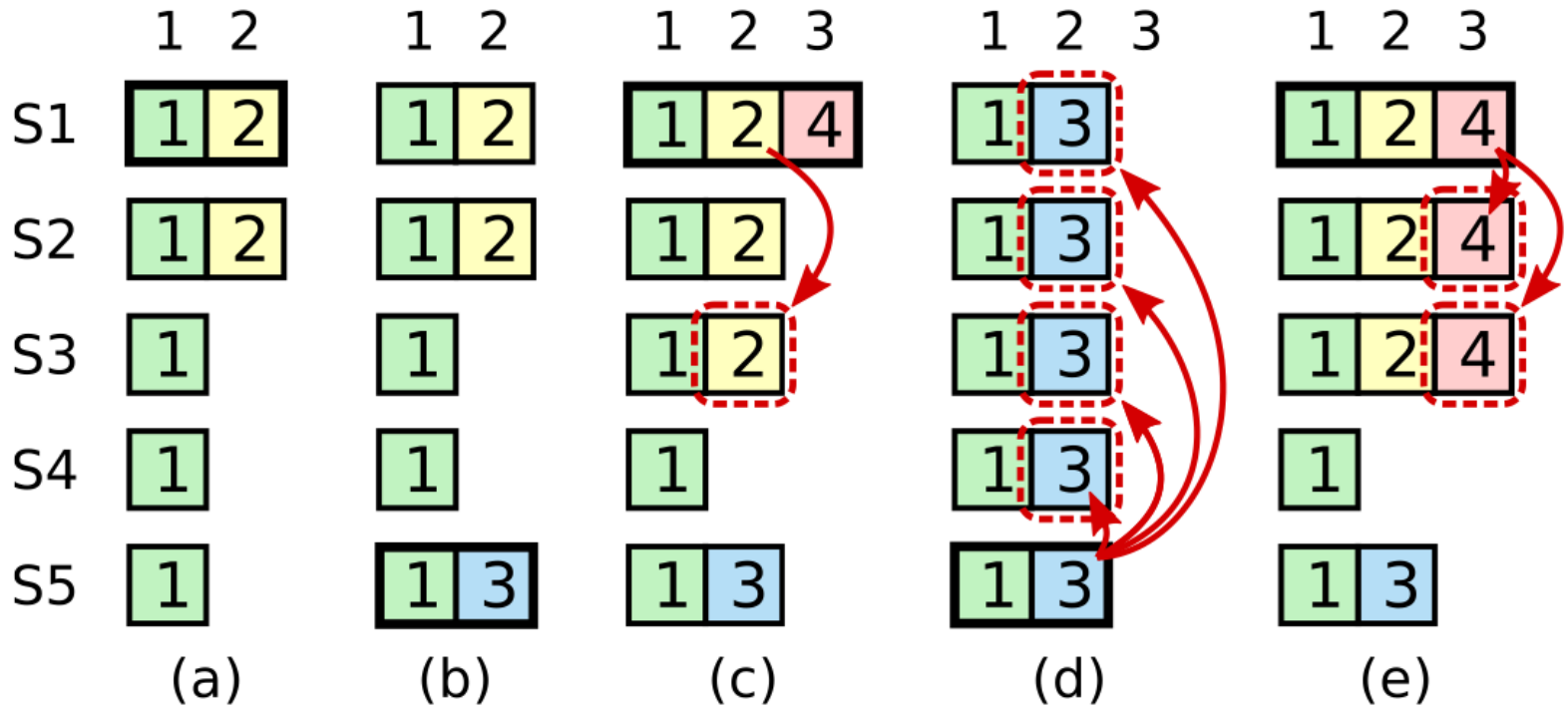
Missing entries → (a-b)

Extra uncommitted entries → (c-d)

Missing + Extra uncommitted entries → (e-f)



Committing Entries From Previous Terms





Why Mutual Exclusion?

Balance = Balance + 1 equivalent in assembly

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

Want 3 instructions to execute as an uninterruptable group

- Want them to be atomic; appears that all execute at once, or none execute

Uninterruptable group of code is called critical section

More general: Need mutual exclusion for critical sections

- If thread A is in critical section C, thread B isn't
- It is fine if other threads do unrelated work

Distributed Locking

Cannot share local lock variables

How do we support mutual exclusion in a distributed system?

Let us start simple with a central solution

Distributed Lock – Central Solution

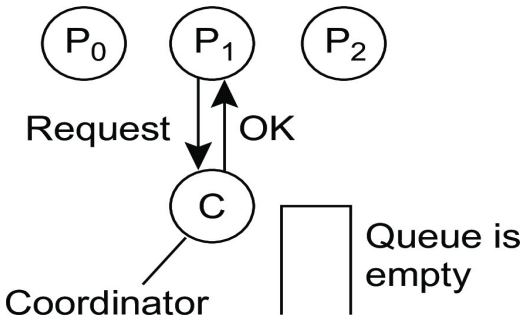
System Model

- Each pair of nodes is connected by reliable channels
- Messages are eventually delivered to recipient, and in FIFO order
- Nodes do not fail

Central Solution

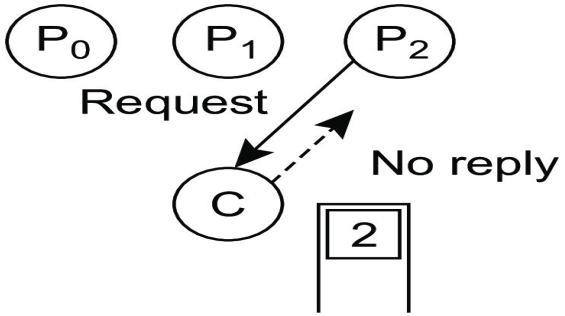
- Elect a central leader using election algorithm
- Leader keeps a queue of waiting requests from nodes who wish to access CS

Distributed Lock – Central Solution (contd…)



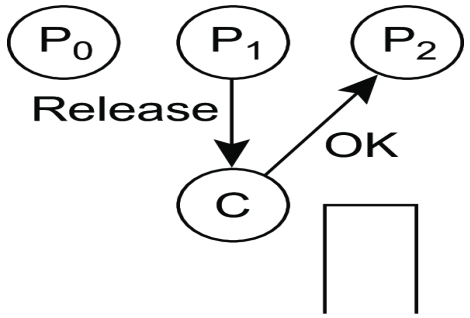
Step 1

Node P1 asks the coordinator for permission to access a shared resource. Permission is granted.



Step 2

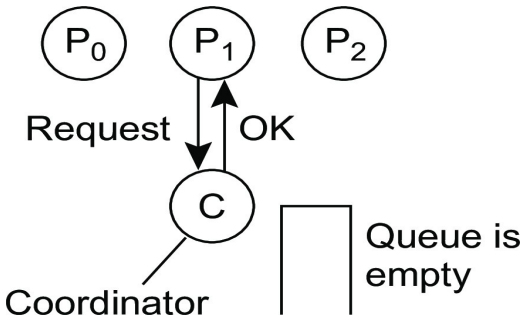
Node P2 then asks permission to access the same resource. The Coordinator does not reply.



Step 3

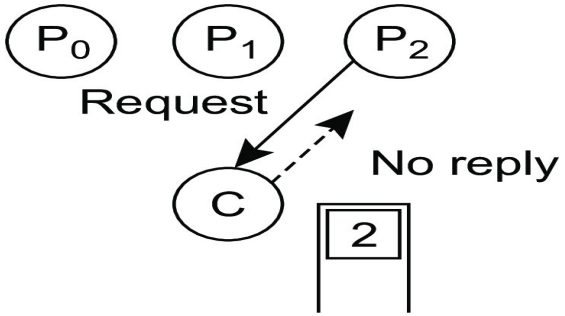
When P1 releases the resource, it tells the coordinator, which then replies to P2.

Distributed Lock – Central Solution (contd…)



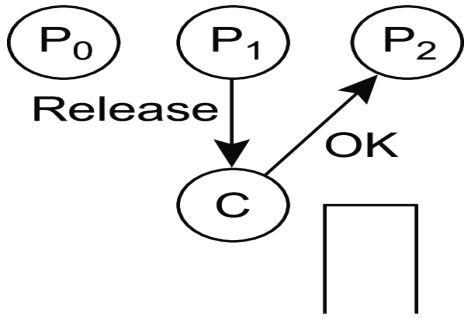
Step 1

Node P1 asks the coordinator for permission to access a shared resource. Permission is granted.



Step 2

Node P2 then asks permission to access the same resource. The Coordinator does not reply.



Step 3

When P1 releases the resource, it tells the coordinator, which then replies to P2.

Problems????

Distributed Solution

Decentralized approach

- All nodes involved in the decision making of who should access the resource

Ricart-Agarwala Algorithm

Use the notion of causality – rely on logical timestamps

Ricart-Agrawala Algorithm

Requestor

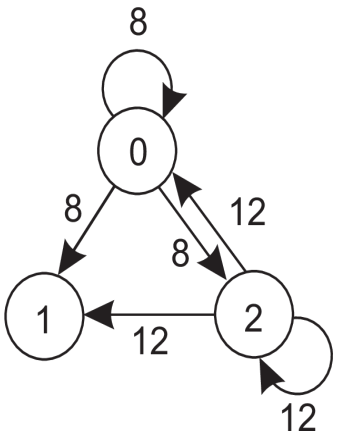
1. Broadcast a message to all receiver (including itself)
   <Resource-Name, Node-Name, Logical Timestamp>

3. Wait for all the OK messages.
4. Access resources once all receivers send OK message.
5. Release the resource; Send OK message to all queue entries

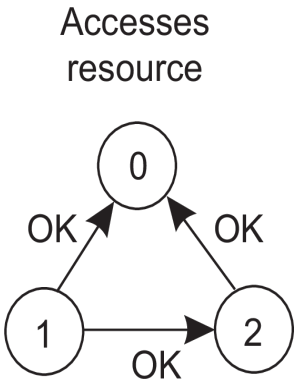
Receiver

2. If receiver not accessing the resource or does not want to access it, send OK message to the sender.
   If the receiver already has access to the resource. Do not reply. Queue the request.
   If receiver wants to access the resource but has not yet done, compare the timestamp
   If incoming message has lower timestamp:
   send OK message to the sender
   Else:
   Queue the incoming request and send nothing

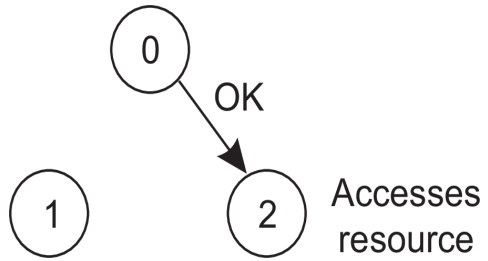
Ricart-Agrawala Algorithm Example



Step 1



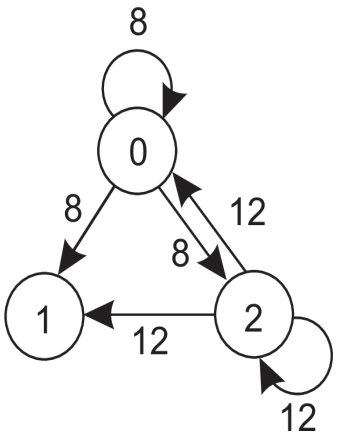
Step 2



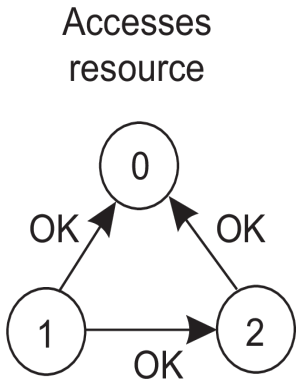
Step 3

Step 1: Two nodes want to access a shared resource at the same moment.
Step 2: P0 has the lowest timestamp, so it wins.
Step 3: When process P0 is done, it sends an OK message to P2. P2 can access the resource thereafter.

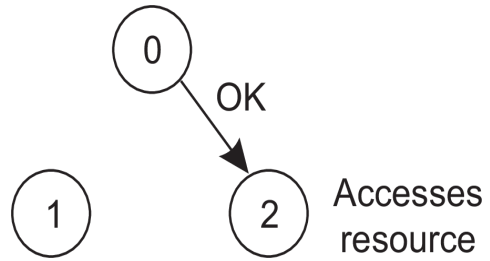
Ricart-Agrawala Algorithm Example



Step 1



Step 2



Step 3

Step 1: Two nodes want to access a shared resource at the same moment.
Step 2: P0 has the lowest timestamp, so it wins.
Step 3: When process P0 is done, it sends an OK message to P2. P2 can access the resource thereafter.

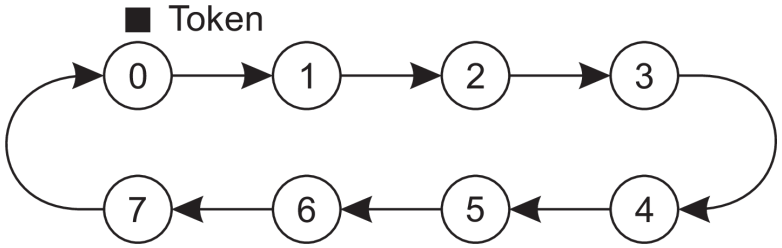
Problems????

Token Ring Algorithm

All nodes arranged in a ring fashion

Use token as a means of ownership

- Whosoever has the token can access the resource
- If no access needed, pass it on to the neighbor
- Token gets passed to all the nodes



Problems????

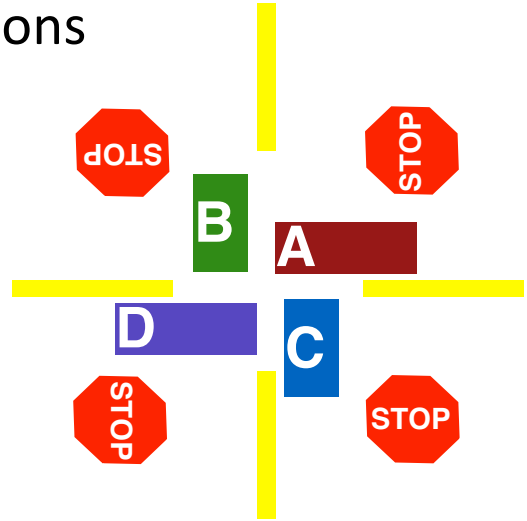
Deadlocks

No progress can be made because two or more nodes are each waiting for another to take some action and thus each never does

Deadlocks can only happen with these four conditions

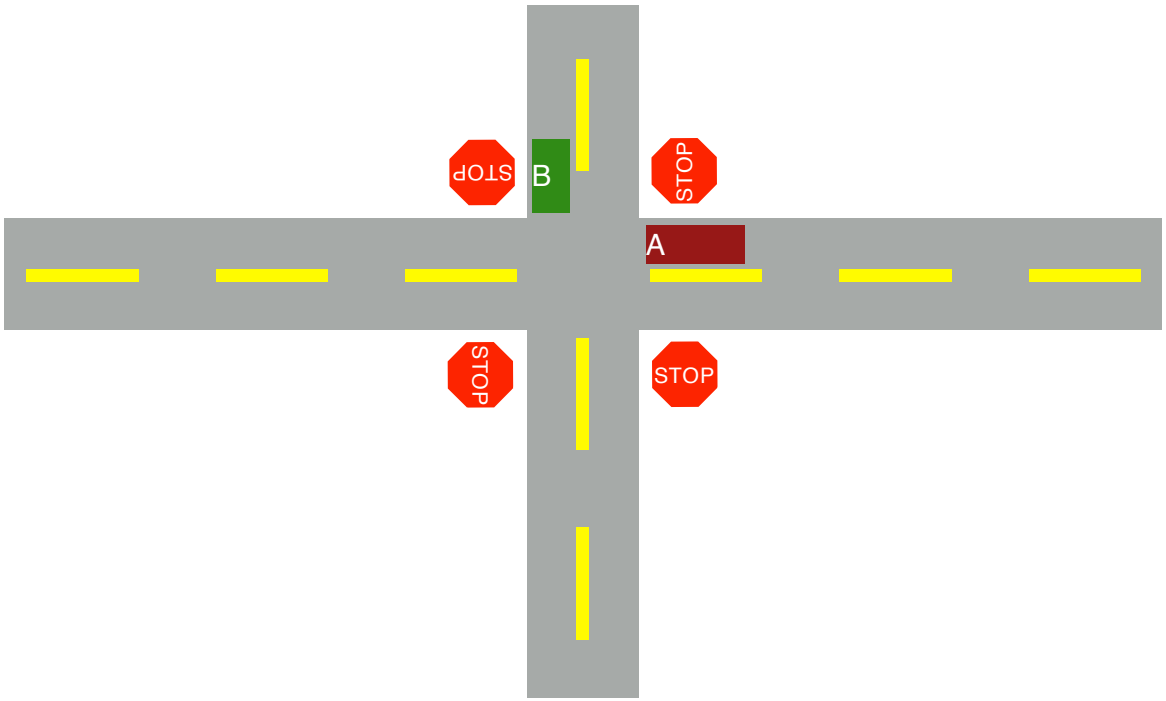
1. Mutual Exclusion
2. Hold-and-wait
3. No preemption
4. Circular Wait

Leads to safety property violation



Deadlock Example – Real World Case

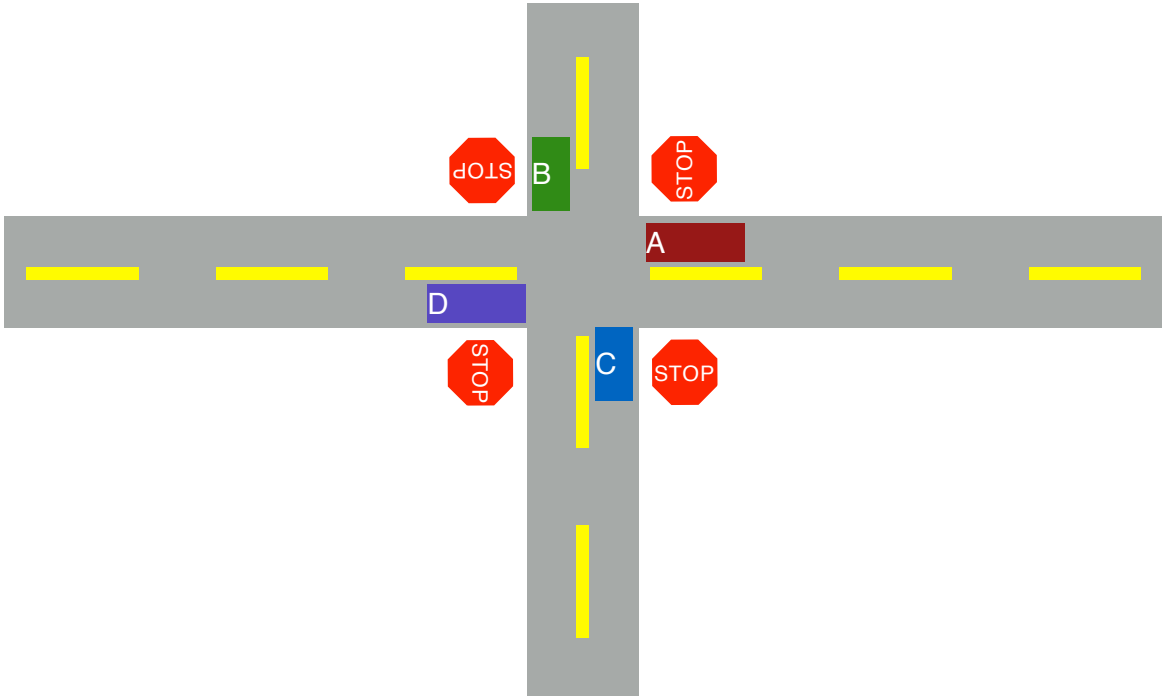
Both cars arrive at same time
Is this deadlocked?



Conditions necessary for a deadlock:
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Deadlock Example – Real World Case (contd..)

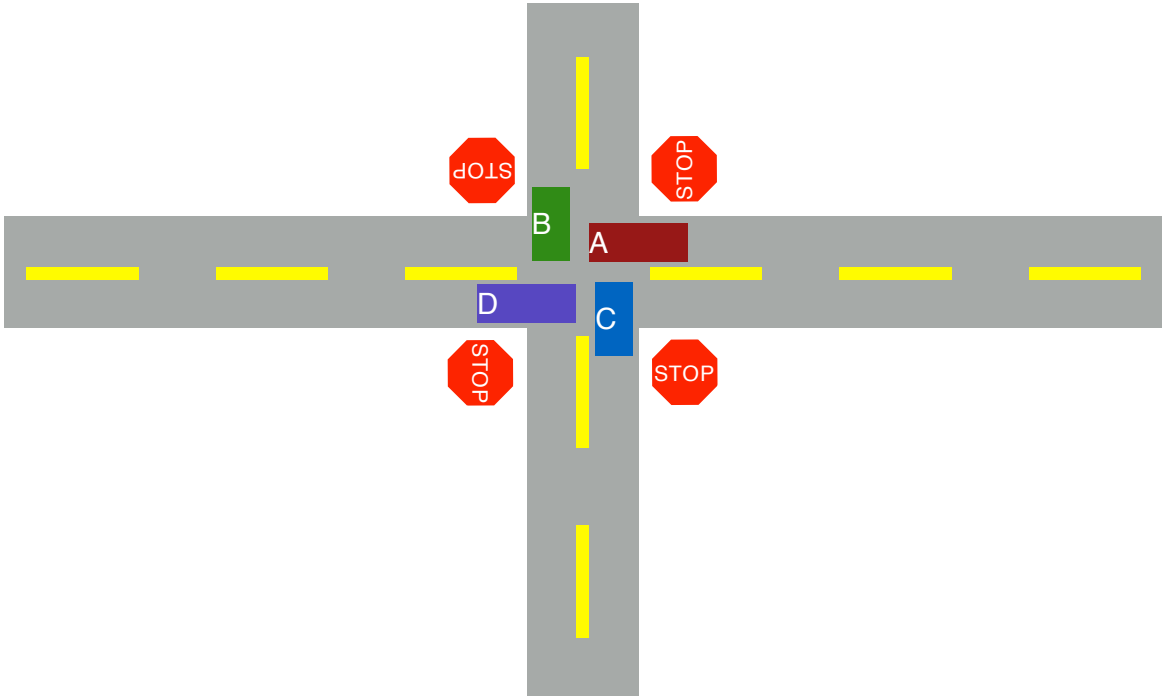
4 cars arrive at same time
Is this deadlocked?



Conditions necessary for a deadlock:
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Deadlock Example – Real World Case (contd..)

4 cars arrive at same time
Is this deadlocked?

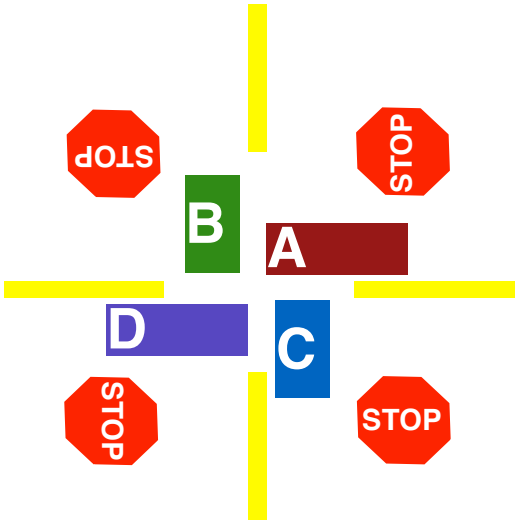


Conditions necessary for a deadlock:
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Deadlocks Handling

Two main strategies

- Prevention – Eliminate any one condition
- Detect and Handle



Eliminate Hold-And-Wait Condition

Problem: Nodes hold resources while waiting for additional resources

Deadlock Prevention Strategy

- Acquire all the locks atomically
- Can release locks over time, but cannot acquire again until all locks have been released

How?

- Use a meta lock

Eliminate No Preemption Condition

Problem: Locks cannot be forcibly removed from nodes that are held

Strategy: If thread can't get what it wants, release what it holds

```
top:
        lock(A);
        if (trylock (B) == -1) {
                unlock(A);
                goto top;
        }
```

Eliminate No Preemption Condition (contd…)

Problem: Locks cannot be forcibly removed from nodes that are held

Strategy: If thread can't get what it wants, release what it holds

```
top:
        lock(A);
        if (trylock (B) == -1) {
                unlock(A);
                goto top;
        }
```

Livelock:

No processes make progress, but state of involved processes constantly changes

Classic solution: Exponential random back-off

Eliminate No Preemption Condition

Problem: Locks cannot be forcibly removed from nodes that are held

Strategy: If thread can't get what it wants, release what it holds

```
top:
        lock(A);
        if (trylock (B) == -1) {
                unlock(A);
                goto top;
        }
```

Other strategy: Use timeouts instead of trylock. If lock not acquired within a timeout, release locks

Problem: How long should be the timeout?

Detect Circular Wait Dependency

Another strategy is to detect deadlocks

Find cycles in the wait graphs

- Take snapshots using Global Snapshot Algorithm
- Detect cycles in the snapshot
- Abort tasks to break the cycle


Scan to check-in
Expires at 06/11/2026 18:10