THE UNIVERSITY of EDINBURGH

Distributed Systems Fall 2024

Yuvraj Patel

Today’s Agenda

Leader Election

Mutual Exclusion

- Basics
- Locks

Coursework Discussion

Next Class is on Tuesday(29/10) and not Monday(28/10)

Leader Election Problem

Need to elect leader to perform tasks and broadcast leader details

If leader fails

- Someone will detect leader failed
- Initiate a leader election to elect another leader
- Only one leader elected, and everyone agrees on who is the leader

System Model & Assumptions

System Model

- N nodes in the system; each node having unique id
- Communicate via messages; messages will eventually be delivered
- Failures/crashes may happen at arbitrary time

Assumptions

- Any node can call for an election
- Any node can call for atmost one election at a time
- Multiple processes can call for an election simultaneously; still lead to a single leader
- Result independent of who calls for an election

Bully Algorithm

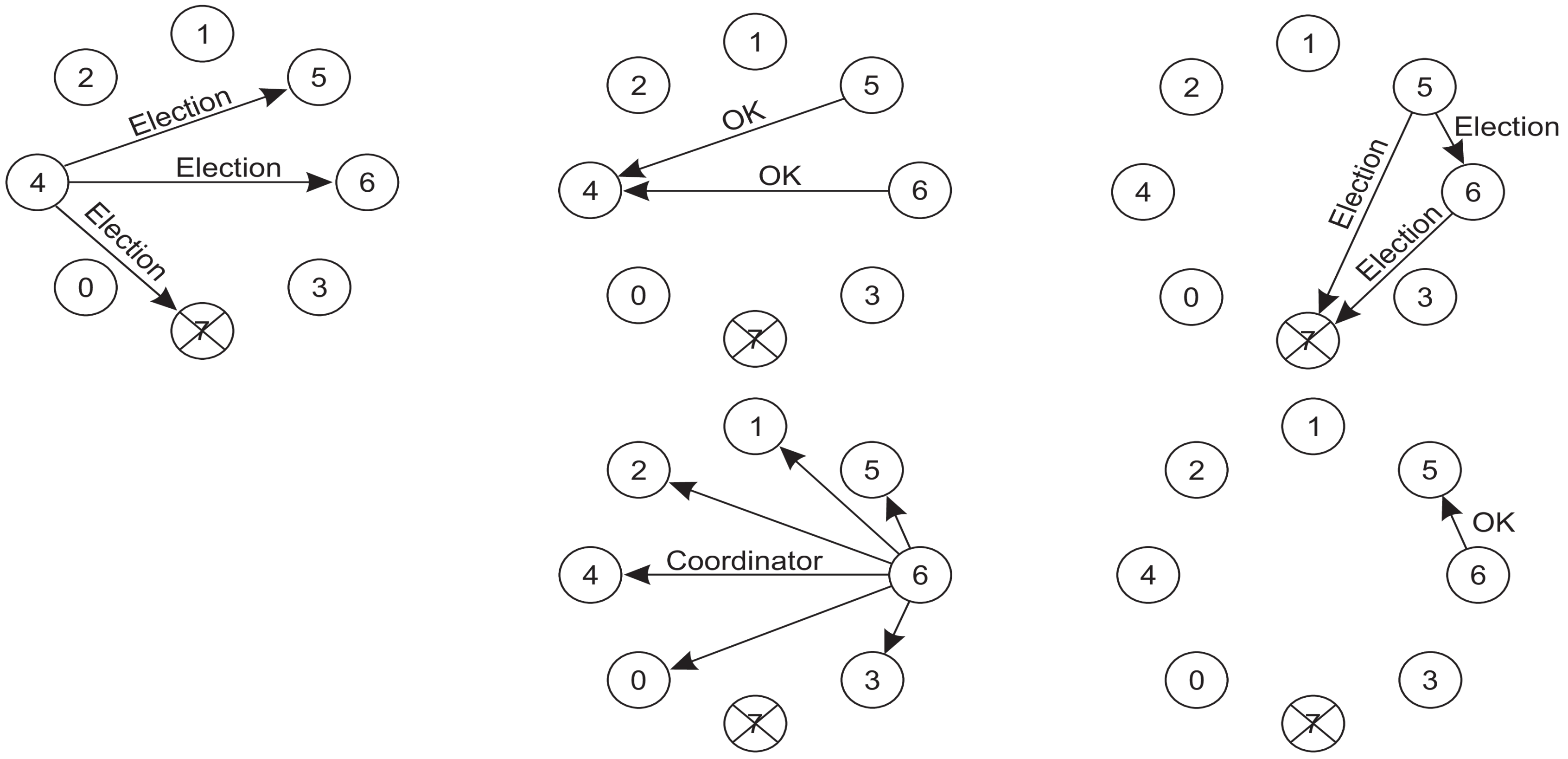
Key Idea: Node with highest ID wins

Consider N nodes {N_0, N_1, $N_{2\ldots}$ N_n}.

Whenever a node N_k notices that the leader is unresponsive, election initiated

- N_k sends an ELECTION message to all the processes with higher IDs: N_{k+1},… N_n
- If no one responds, N_k wins
- If one of the higher-up's answers, it takes over and N_k's job is done

Example



Election
Election
Election
OK
OK
Election
Election
Election
Coordinator
OK

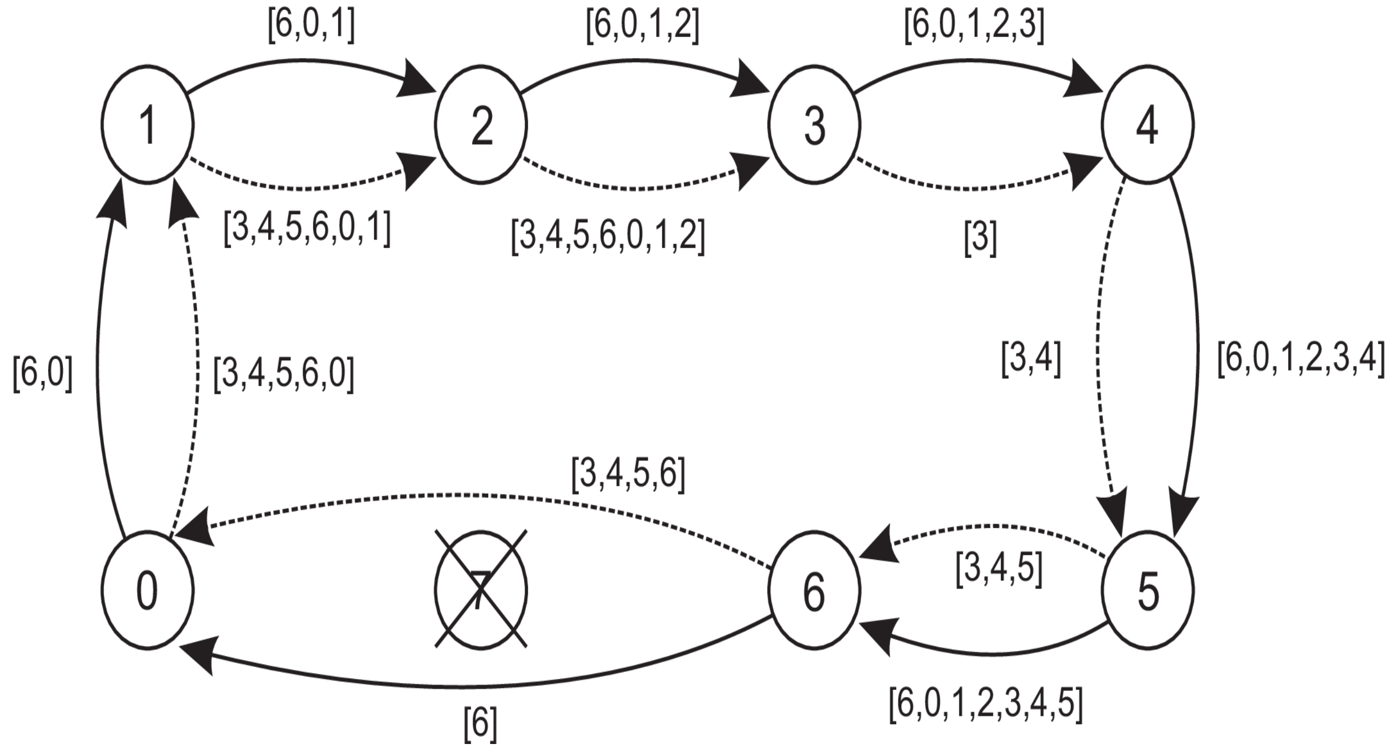
Ring Algorithm

Nodes are organized into a ring. Process with highest id is elected as coordinator

Whenever a node N_k notices that the leader is unresponsive, election initiated

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on the next successor
- If a message is passed on, the sender adds itself to the list.
- When the message gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all the living nodes. The one with the highest id is elected as coordinator

Example



The solid line shows the election messages initiated by N_6
The dashed one is election messages initiated by P_3

Both have the same list so it is safe to have two nodes initiating an election

Concurrent Executions

```
balance = balance + 1; balance at 0x9cd4 = 100
```

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

HW State / After Instr	T1: 0	T1: 1	T1: 2	T1: 3	T2: 0	T2: 1	T2:2	T2:3
0x9cd4								
%eax								
%rip								

thread control blocks:

Thread 1	Thread 2
%eax:	%eax:
%rip:	%rip:

Concurrent Executions (contd…)

```
balance = balance + 1; balance at 0x9cd4 = 100
```

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

HW State / After Instr	T1: 0	T1: 1	T1: 2	T2: 0	T2: 1	T2:2	T2:3	T1: 3
0x9cd4								
%eax								
%rip								

thread control blocks:

	Thread 1	Thread 2
	%eax:	%eax:
	%rip:	%rip:

Non-Determinism

Concurrency leads to non-deterministic behavior

- Different results even with same inputs

Race conditions: Specific type of bug

- Sometimes program works fine, sometimes it doesn’t; depends on timing

Why Mutual Exclusion?

```
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

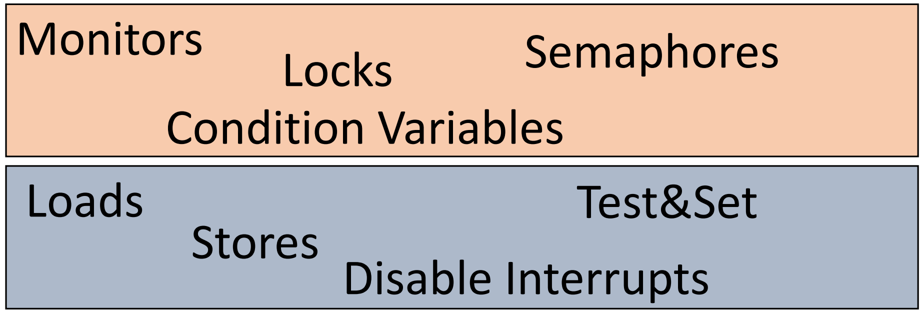
Synchronization

Build higher-level synchronization primitives

Operations that ensure atomic execution and correct ordering of instructions across threads

Use help from hardware

Locks are the commonly used Synchronization primitives



Motivation: Build them once and get them right

Spinlocks

Goal: Provide mutual exclusion

Allocate and Initialize

- pthread_spinlock_t lock; pthread_spin_init()

Lock() – Acquire the lock for exclusive access to the critical section

- Wait (Spin) until the lock is not available (some other process is holding the lock)
- pthread_ spin_lock()

Unlock() – Release the lock and relinquish the exclusive access

- Let another process access the lock and enter the critical section
- pthread_spin_unlock()

Timeline with Spinlock

Thread 1 | Thread 2 | Balance = 0 (initial value)

```
spin_lock()          spin_lock()
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
spin_unlock()
                     mov 0x123, %eax
                     add %0x1, %eax
                     mov %eax, 0x123
                     spin_lock()
```

Correct answer: Balance = 2

Two main properties

- Safety – Nothing bad happens in the system; atomicity needs to be guaranteed; No deadlocks
- Liveness – Eventually something good will happen; atleast one thread will acquire the lock and make forward progress

Distributed Systems

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

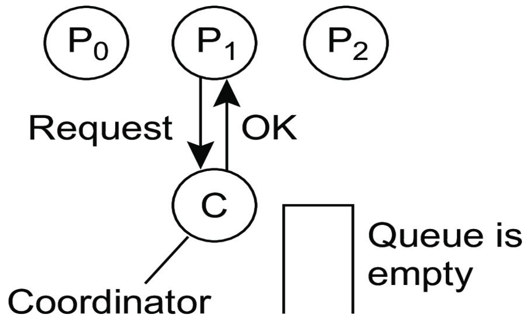
System Model

- 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

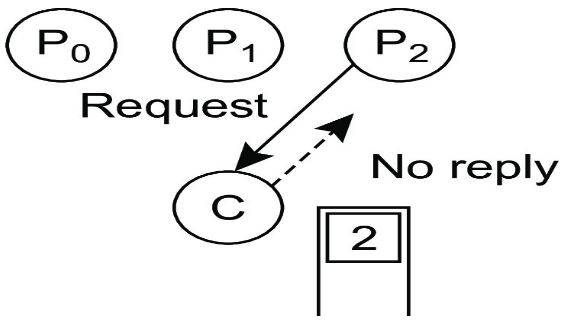
- Elect a central leader using election algorithm
- Leader keeps a queue of waiting requests from nodes who wish to access CS

Distributed Lock – Central Solution (contd…)



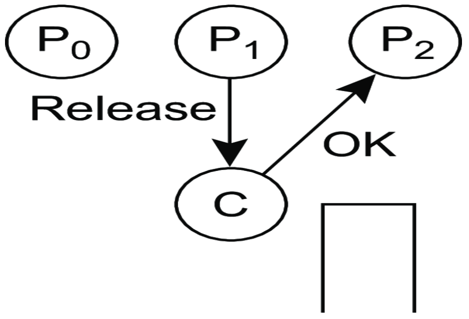
Step 1

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



Step 2

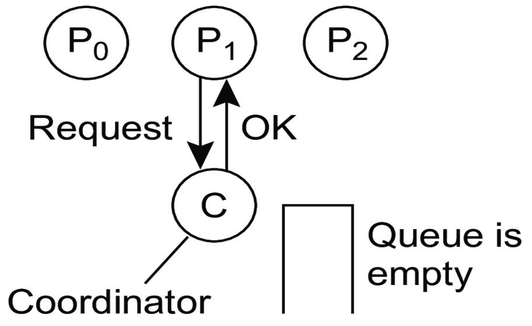
Node P2 then asks permission to access the same resource. The Coordinator does not reply.



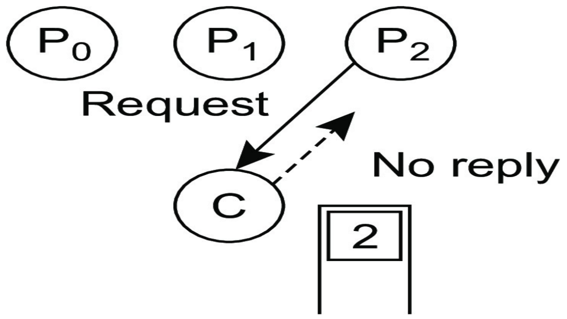
Step 3

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

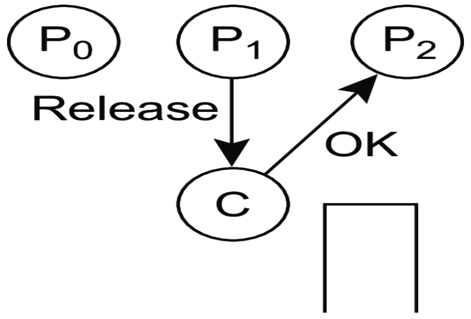
Distributed Lock – Central Solution (contd…)



Step 1



Step 2



Step 3

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

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

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

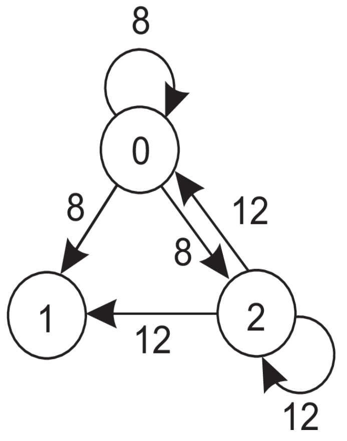
1. Broadcast a message to all receiver (including itself)
   <Resource-Name, Node-Name, Logical Timestamp>

Receiver

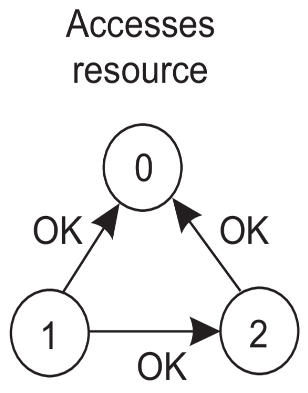
```
2. If receiver not accessing the resource or does not
   want to access it, send OK message to the sender.
   If the receiver already has access to the resource. Do
   not reply. Queue the request.
   If receiver wants to access the resource but has not
   yet done, compare the timestamp
       If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

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

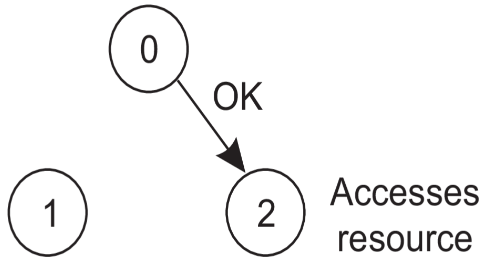
Ricart-Agrawala Algorithm Example



Step 1



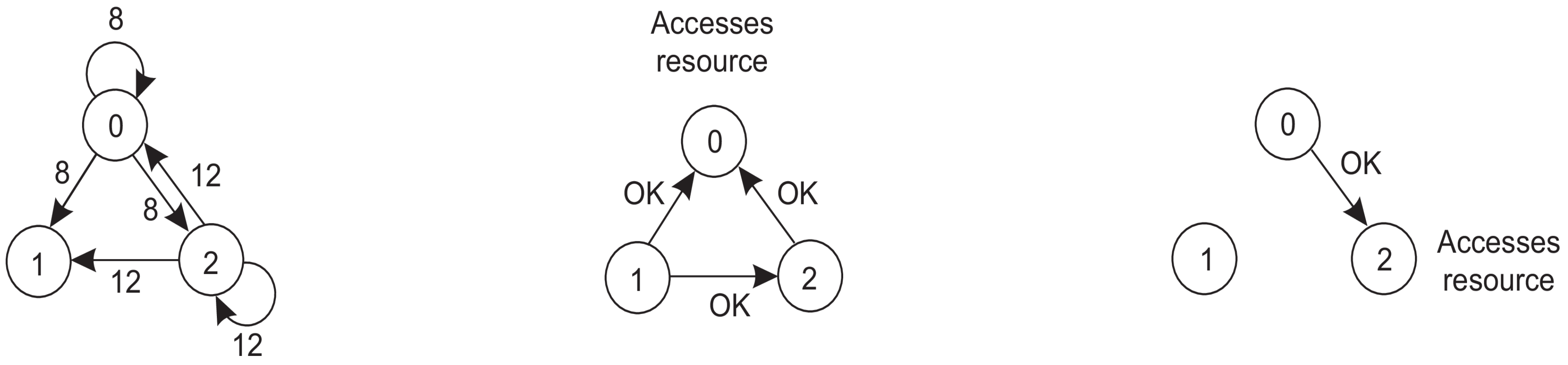
Step 2



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.

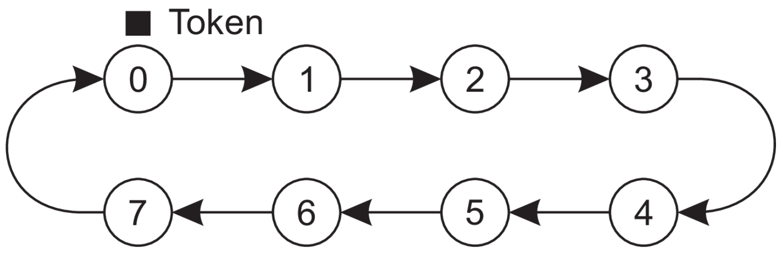
Problems????

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



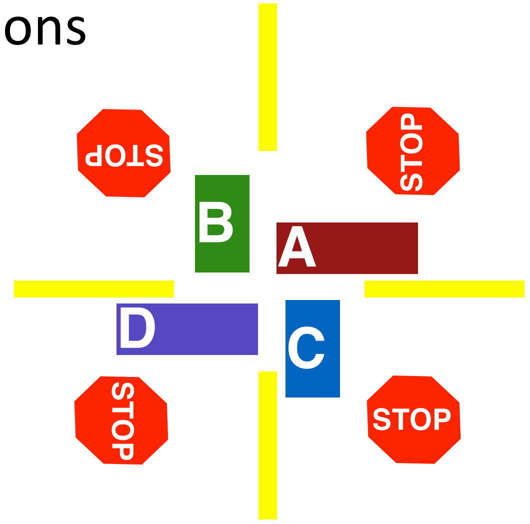
Problems????

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

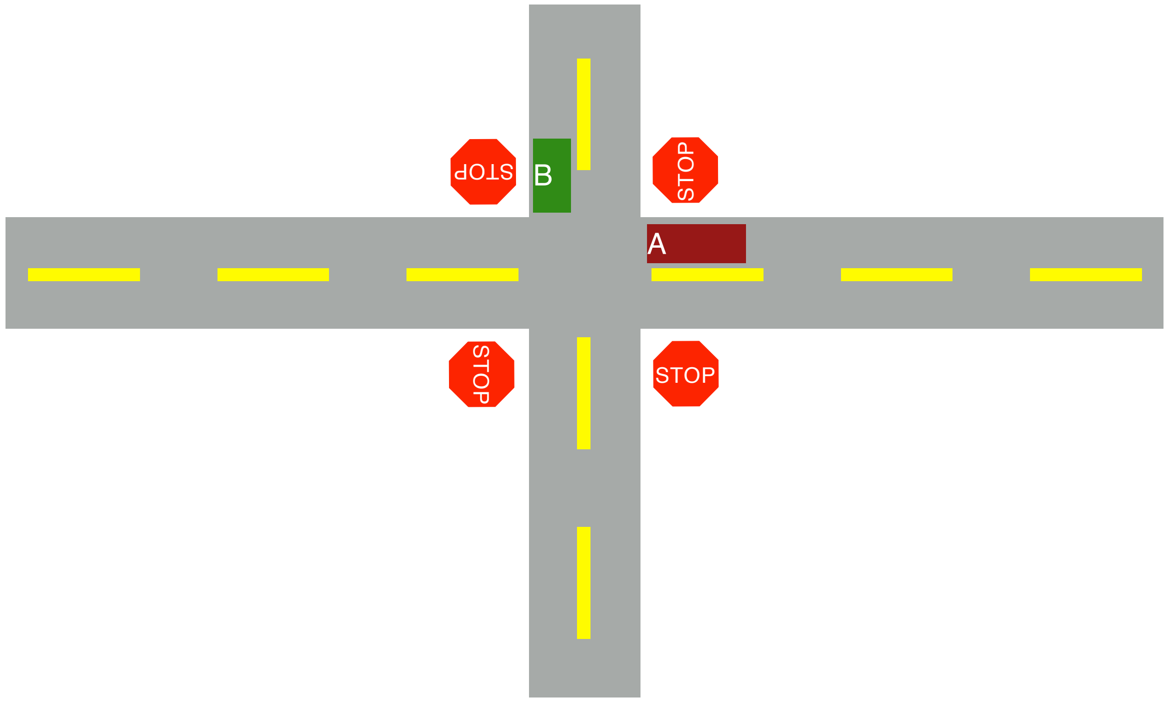
Deadlocks can only happen with these four conditions

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



Deadlock Example – Real World Case

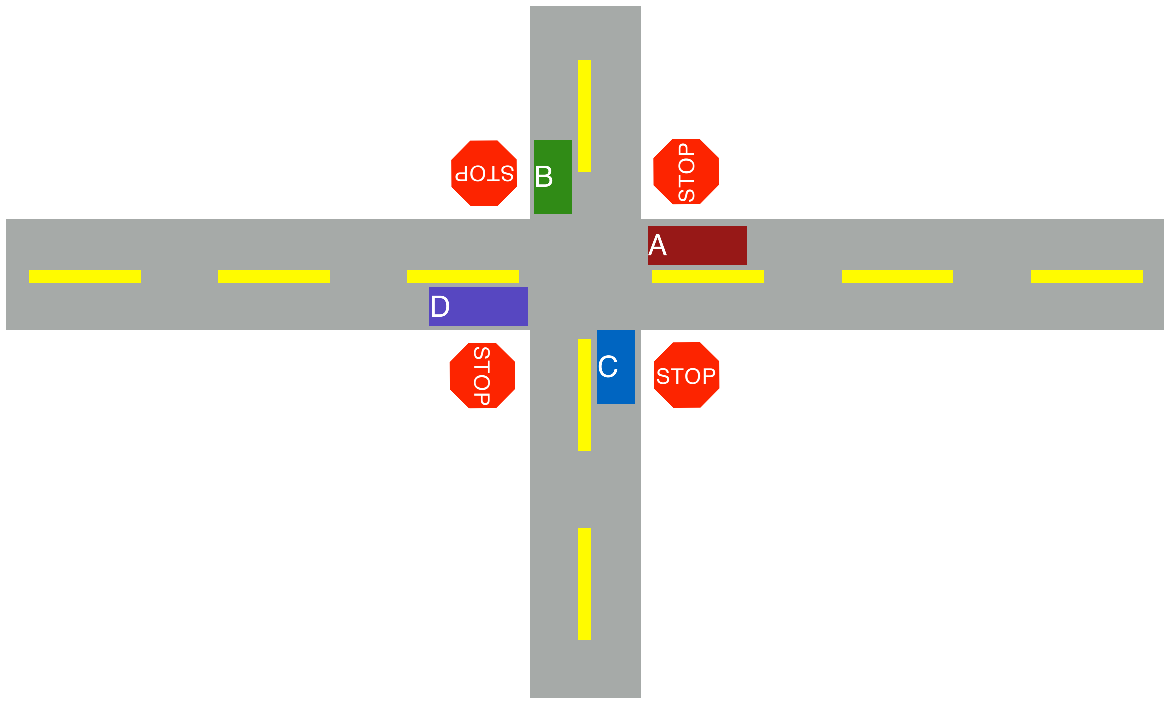
Both cars arrive at same time
Is this deadlocked?



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

Deadlock Example – Real World Case (contd..)

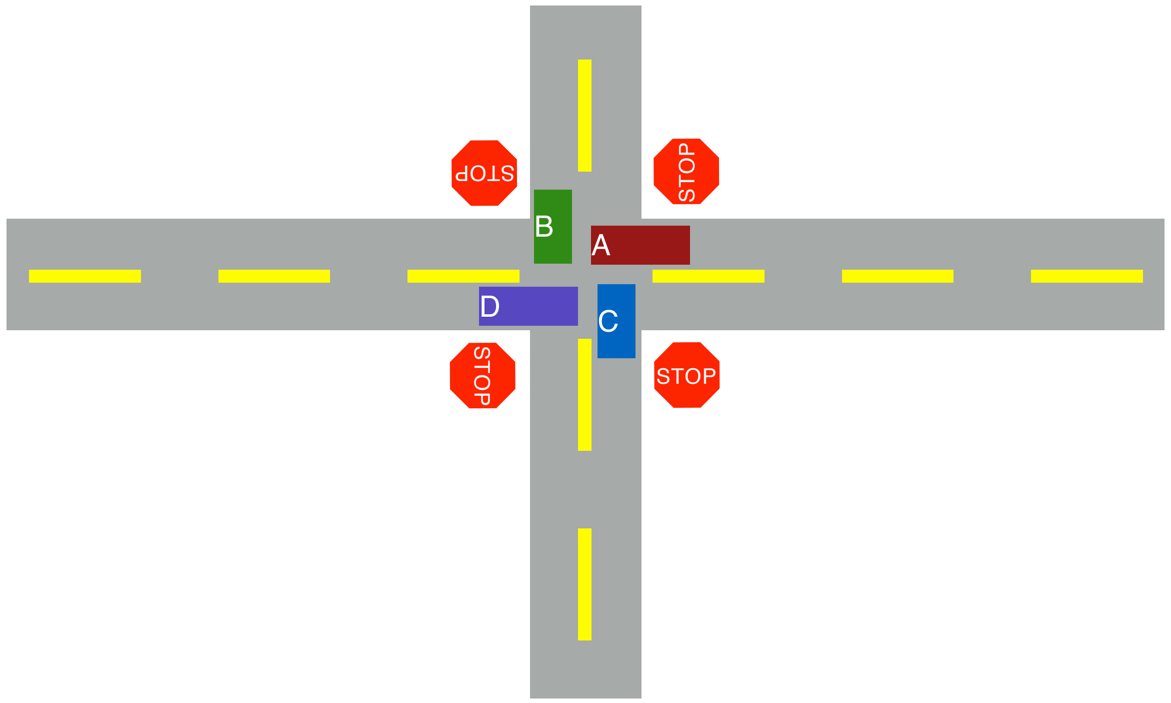
4 cars arrive at same time
Is this deadlocked?



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

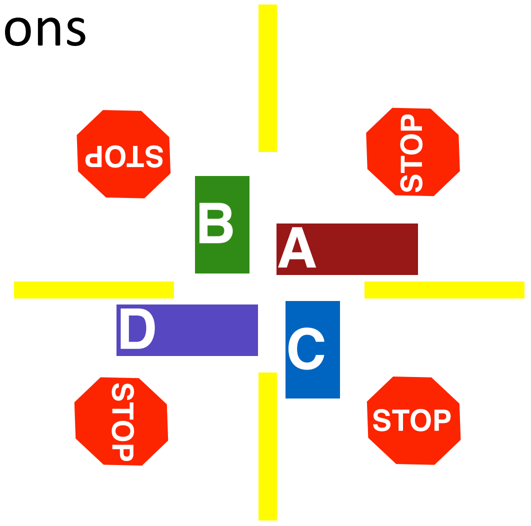
Deadlocks

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

Can eliminate deadlock by eliminating any one condition



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 Hold-And-Wait Condition (contd…)

```
lock(&meta);   lock(&meta);   lock(&meta);
lock(&L1);     lock(&L2);     lock(&L1);
lock(&L2);     lock(&L1);     unlock(&meta);
lock(&L3);     unlock(&meta);
…                             // CS1
unlock(&meta); // CS1         unlock(&L1);
// CS1         unlock(&L1);
unlock(&L1);
// CS 2        // CS2
Unlock(&L2);   Unlock(&L2);
```

Eliminate Hold-And-Wait Condition (contd…)

```
lock(&meta);
lock(&L1);
lock(&L2);
lock(&L3);
…
unlock(&meta);
// CS1
unlock(&L1);
// CS 2
Unlock(&L2);
```

```
lock(&meta);
lock(&L2);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);

// CS2
Unlock(&L2);
```

```
lock(&meta);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);
```

Disadvantages

- Must know ahead of time which locks will be needed
- Must be conservative (acquire any lock possibly needed)
- Degenerates to just having one big lock (reduces concurrency)

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

Coursework Questions