CHALMERS
EXAMINATION / TENTAMEN

Course code/kurskod	Course name/kursnamn			
TDA297	DISTRIBUTED SYSTEMS (ADVANCED COURSE)			
Anonymous code Anonym kod		Examination date Tentamensdatum	Number of pages Antal blad	Grade Betyg
TDA297-31		2015-03-18	17	

Solved task Behandlade uppgifter No/nr		Points per task Poäng på uppgiften	Observe: Areas with bold contour are to completed by the teacher. Anmärkning: Rutor inom bred kontur ifylles av lärare.
1	✓	10	
2	✓	10	
3	✓	10	
4	✓	5	
5	✓	10	
6	✓	9	
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
Total examination points Summa poäng på tentamen		53+3	

QUESTION 1.

a) Give the definitions of Linearizability and Sequential Consistency. Also give examples of application scenarios where each type of consistency may be required.

Sequential consistency: Given a concurrent execution, there exists one sequential execution that produces the same result preserving the order of computations at node level. I.e. each node executes instructions in an internal order, then seq. consistency preserves this order for each node.

Example. Node 1: push(4) pop() → 7

Node 2: push(7)

→ push(4) push(7) pop() → 7

Linearizability: Appart from the sequential consistency constraints, linearizability imposes that real time execution order must be preserved. ✓

(3)

Node 1 Node 2

time ↓

enq(x, 1)

deq(x) → ∅

deq(y) → ∅

enq(y, 2)

→ Sequential consistent

deq(x) → ∅ | deq(y) → ∅ | enq(x, 1) | deq(x) → 1

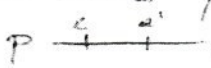
But non-linearizable.

Linearizability is more suitable for banking transactions as it provides higher consistency guarantees, but it is more complex to implement, thus for less critical data replication (e.g. such as Facebook group communication) sequential consistency is used instead.

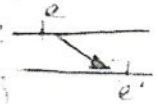
b) What are the differences between causal and total ordering? Also give examples of application scenarios where each type of ordering may be required. (3)

Causal order: Let e, e' be two events, let $e \rightarrow e'$ be the relationship e causally precedes e', defined by:

- If e, e' happened in the same processor, e happened before e'.



- If e, e' happen in two different processors, there is at least one message event from P_i to P_j in the interval between the occurrence of e and the occurrence of e'.



Given this formal definition of causal precedence, we can define causal order as follows:

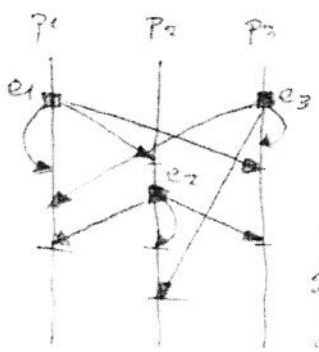
If $e \rightarrow e'$, then every processor in the system will deliver e before e'.

Total order: Let P be a set of processors $\{P_1, P_2, \ldots, P_n\}$. Let e, e' be two events. Total order is defined as follows:

If $\exists i \in \{1, \ldots, n\}$ so that P_i delivers e before e', then $\forall i \in \{1, \ldots, n\}$ P_i will deliver e before e'. In other words, all the processors deliver all events in the same order.

To see the differences between causal and total ordering let's consider two scenarios: a) and b):

a)

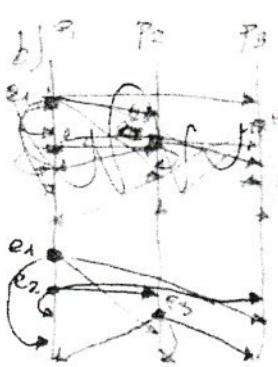


$e_1 \rightarrow e_2$

In all processors, e_1 is delivered before e_2, we have causal order ✓

But if we see e.g. e_1 delivered in different order in all processors, we do not have total order ✗

b)



$e_1 \rightarrow e_2$

but e_1 is delivered after e_2 in P_3. We do not have causal order

However, they are still delivered in same order in all processors so we have total order

QUESTION 1 - b) Continues.

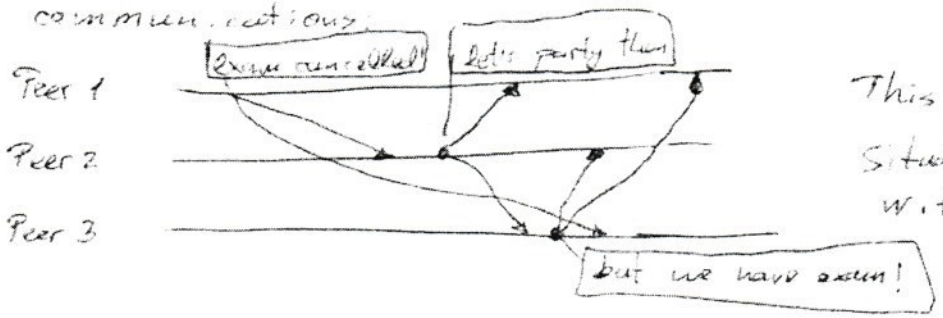
So as we can see, because of a) Causal order $\not\Rightarrow$ Total order and because of b) Total order $\Rightarrow$ Causal order. Thus:

Causal order $\not\Leftrightarrow$ Total Order.

Which means that they are two different ordering registers. However they can be combined under certain conditions to provide causal and total ordering which have the benefits of both orderings.

Application scenarios:

- Causal ordering is useful for group communication (for example a group in whatsapp) to avoid incoherent communications.

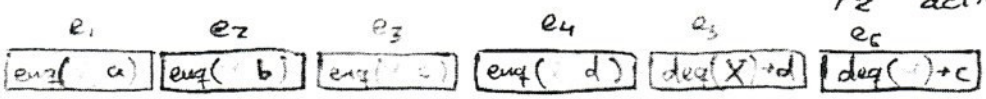


This kind of situation is avoided with causal ordering

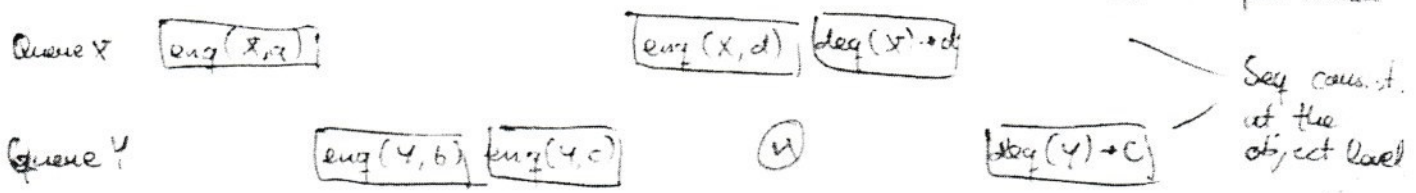
- Total ordering is used in replication to ensure that all the replica managers (RM) has the same data. (In the same order (All updates performed)). However, this ordering is rarely used in highly critical replication, together with other orderings to ensure consistency. For example, in banking systems.

c) A correctness property P is compositional if, whenever each object in the system satisfies P, the system as a whole satisfies P. Is sequential consistency compositional? If yes explain why, if no give an example.

No. Let's consider this example. P_1 actions P_2 actions



If $e_5 \rightarrow e_2$ then $e_3 \rightarrow e_2$, thus, sequential consistency is not compositional



QUESTION 2

a) What is meant by Quorum-based reads and writes? Can one increase only the availability of reads?

Quorum-based reads and writes is based in the idea of grouping the replica managers in quorums, which are sub-groups of RMs that satisfies a series of conditions defined in detail in the Gifford quorum algorithm so that read quorums always have at least one up-to-date RM and write quorums always intersect with both all write quorums and all read quorums. This way we do not have to either update the data in all RM in one update operation nor read data from all RM in one read operation, achieving this way eventual read consystency but virtual consystency at consultation time in a more efficient way than updating all the replicas.

Of course one can increase only the availability of reads by modifying the size of the read quorum and tolerating the failure of one or more nodes in it. This would also affect the write quorum as votes need to be shared among the RMs in a careful way to ensure that even with nodes failing in a read quorum we will always get an up-to-date RM.

(2) b) What type of consistency does a Quorum based system guarantee and How? Explain your answer.

It ~~gu~~ guarantees eventual sequential and linearizable consistency by ensuring the non-empty intersections of readand write quorums and using version numbers associated to the replicated data so that the correct order of events can be always constructed and updated thus ensuring update operations in the correct order and read operations receiving data up to date with the order established by the updates.

(3) c) GFS istance with 3 replicas for each block of a large file. Erik claims that GFS can restore full redundancy faster than with traditional 3-way replication. Is he right? Explain your answer.

Matches with intuition

Yes. He is right. Given that the sets of servers do not contain identical sets of blocks, to restore full redundancy the GFS system ~~can take advantage of faster local reads than entirely getting data through the network from and it's cause load to after be allowed to temporarily store more than one replica and the work load to later restore it~~ can more flexibly distribute the workload of reconstructing a replica among all processors in the system. Thus we are not forced to setup a dedicated server which would take data from only 2 other servers which would work at peak load during the take-over.

It means that in CFS, replica managers information can be merged using a quorum based approach so that we can more flexibly restore full redundancy.

(3) d) Implementation of a replicated distributed priority queue. Insert and DeleteMin operations supported. Tolerating 2 replica crashes for both operations. Taking care of cleaning memory.

In order to tolerate 2 replica crashes, I am going to consider a quorum based read-write algorithm using 5 replicas all of them with a vote 1 according to Gifford algorithm. Thus the size of the write quorum will be 3 and of the read quorum will be 3. ✓

In order to manage memory I will consider logs of instructions including only the Insert operations in the logs and implicitly performing the DeleteMin operation by removing the corresponding insert operation. The logs will have a vector timestamp ✓ associated so that the merge operation can be applied when reading and writing thus having all the replicas providing a consistent service.

Deleting the corresponding insert operation log whenever a DeleteMin operation is realized will ensure the cleaning of all unnecessary memory.

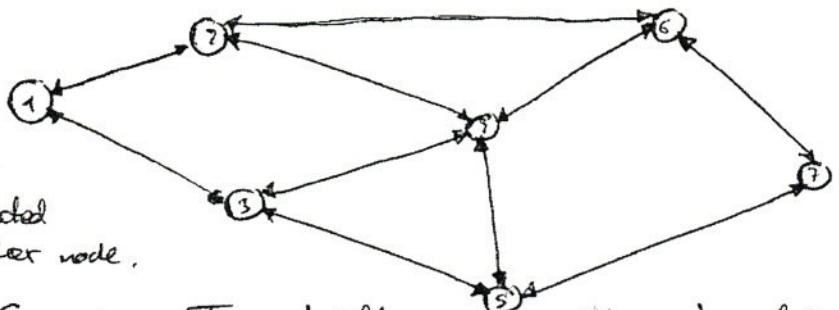
To define the depends on relation between insert and deleteMin we have to ensure that: (next page)

Logs of insertions are sorted by value (not by timestamp)
Global timestamp of a log which is the latest timestamp of insert operations in the system is immediately before ~~the~~ timestamp of a deleteMin operation.
When this situation occurs, which is guaranteed in the quorums by Clifford, the head of the log (element with min value) is deleted from the log and the global vector timestamp of the log is updated accordingly.
So, deleteMin will remove from the log the insertion register associated to the smallest value in the system.
It is also necessary to deal with log merging, and this can be doing applying ~~the~~ a variation of merge sort in the logs in the quorum.
With all this we have covered the update and cleaning phase all in once.

PROBLEM 3

a) Algorithm for leader election on an arbitrary topology.

Fig 1: Arbitrary topology ☆ Not all nodes need to be connected to every other node.



Idea: Spanning-Tree building algorithm based in broadcast with ACK → Initiator broadcast its id and nodes compare and broadcast id and initiator id, ack are sent back initiator gets all ack → every node has sent message till end of graph spanning tree is built, id of the leader is returned with the ack. To the initiator, it can then use the spanning tree to broadcast it. Other initiators → Reuse spanning tree or contribute to building it.

Protocol: Let's first assume one initiator and later on expand the solution. Let's also assume all nodes start without a parent node defined.

(4)

Initiator code:

```
Set ack = {n : for n ∈ N : neighbourhood of i}
Set son set = {};
initiation { for all neighbors n:
    send ( Re. initiation msg with my id );
    ack[n] = 0;
}
counter = 0, candidate id = my id;
While (counter != N){
ack reception and process {
    receive(ack from neighbor);
    check leader candidate id in ack;
    update leader candidate, if ack from
    counter++;
    if(ack brings son) update son set to sender
    ack[n] = 1;
}
Set leader = candidate;
for sons in sons{
    Send message informing about leader id
                (candidate after all ack received)
}}
```

Node code:

```
On Re. initiation msg receptio
    ack = {s : me*};
    son set = {};
    if (first time receiving this msg)
        parent = sender;
    } else {
        respond with last ack;
        refraining min son;
        return;
    }
    Compare candidate id with
    my id, update candi. id
    for all neighbors but parent
    send msg with candi id.
    wait all acks
    on ack from son
    replay parent with ack
    updated from sons
}
On leader elected message {
    update leader id
    send to all sons
}
```

So we are using acks to build son-parent relationship in graph thus building a spanning tree, and inform about the candidate id in our subtree which will let the initiator have all the candidate ids aggregated thus knowing who is the leader and being able to propagate this info using now the spanning tree.

In case another node initiates the election, it will either reuse the spanning tree or built a new one until it meets the other one and then both trees can be either merged or maintained separately for each candidate. The first option is more complex the second one cost. more in memory (one spanning tree per initiator node). But same info will finally reach all nodes.

b) Complexity and com. complexity. $G=\{V, E\}$ edges = communication channels, vertex = nodes

First round of broadcast:

All node sends/receive one message per edge in only one direction, so com complexity of this round is $|E|$. ✓ (3)

Time complexity is $O(D)$ D being the diameter of the graph: Each new node sending a message is one step closer to the furthest node. However, for async systems we can be unlucky and have a trivial tree of height $|V|$. So complexity $|V|$.

All acks: Same reasoning: Comm complexity $O(|E|)$ time complexity $O(|V|)$

Final round: Spaning tree usage to broadcast leader id: Com complex $O(|V|)$ because only the sp.tree edges are used. Time complexity $O(|V|)$ for the same reasoning. ✱

Conclusion: Overall: Com. complex $O(|E|)$, time complex $O(|V|)$

✱ In synchronous systems time complexity can be reduced to $O(D)$. Third index for synch systems.

c) Is it possible to design a symmetric algorithm for leader election? If your answer is no provide a proof.

No. It is not possible to design a symmetric algorithm for leader election.

Proof by contradiction. Let's assume that all nodes in the system are equall and they run the same i.e. algorithm which let's us model them as a state machine in which each step consists of communication and computation. Let's assume that all nodes start in a state, in which no one knows the leader yet.

After the first round two things can happen: 1:- All the nodes stay in the same state => further rounds are needed. (3)

2:- All the nodes change state and, as all of them have followed the same set of com. operations, all of them transition to the same state:

2.1.- If the new state is leading, the alg. failed because one and only one node can be the leader.

2.2.- If the state is not leading, the alg. failed also because when all nodes have converged to a decision, at least one of them should be the leader.

So, case 2 would fail, thus case 1 will happen, leading again to the same situation. Which, by induction, round n of indecision => round n+1 of indecision will result in an infinite non-converging loop, thus invalidating the algorithm. So we can conclude that no such algorithm exist. □

PROBLEM 4

a) Relation between atomic broadcast and agreement?

Atomic broadcast and agreement have a deep relation together: In case we had an atomic broadcast algorithm we could easily solve agreement by atomically broadcasting the decision of the commander (node sharing the initial order or message).

This relationship have strong implications in the case of asynchronous systems. The result FLP proves that in an asynchronous system with one only processor crashing, it is impossible to reach agreement. Which derives into the following theorem:

Atomic Broadcast is impossible to be reached in asynchronous systems.

Proof. By contradiction Let S be an async system, let BR be an atomic broadcast protocol working on that system. Then, we can use BR to build CR an agreement protocol in the system. However, by FLP, CR can not exists, thus, BR can not exist □.

b) Connection between Atomic Broadcast and the Eager State Machine replication mechanism?

The connection is that the first one is necessary for the second one to be able to work. The Eager State Machine rep. protocol uses atomic broadcast each time a Replica Manager (RM) receives an update from the Front End (FE) so that all (RM) perform the updates in the same order thus guaranteeing that maintain all the same state in the state machine model.

QUESTION 5

The dining philosophers. All philosophers seek their right fork first.

Does this solve the problem?

No, it does not solve the problem. Let's study the following counter example: Let P be the set of n philosophers in the table, $P = \{p_0, p_1, \ldots, p_{n-1}\}$ with the corresponding set of forks $F = \{f_0, \ldots, f_{n-1}\}$. Let's assume p_0 sits between p_{n-1} and p_1 and p_i is found between p_{i-1} and p_{i+1} and let's assume the following chain of events:

0.- time: All philosophers are hungry and no one is holding any fork.

1.- p_0 pick its right fork (f_0, which was free)

2.- p_1 pick its right fork (f_1, which was free)

3.- p_2 pick its right fork (f_2, which was free)

...

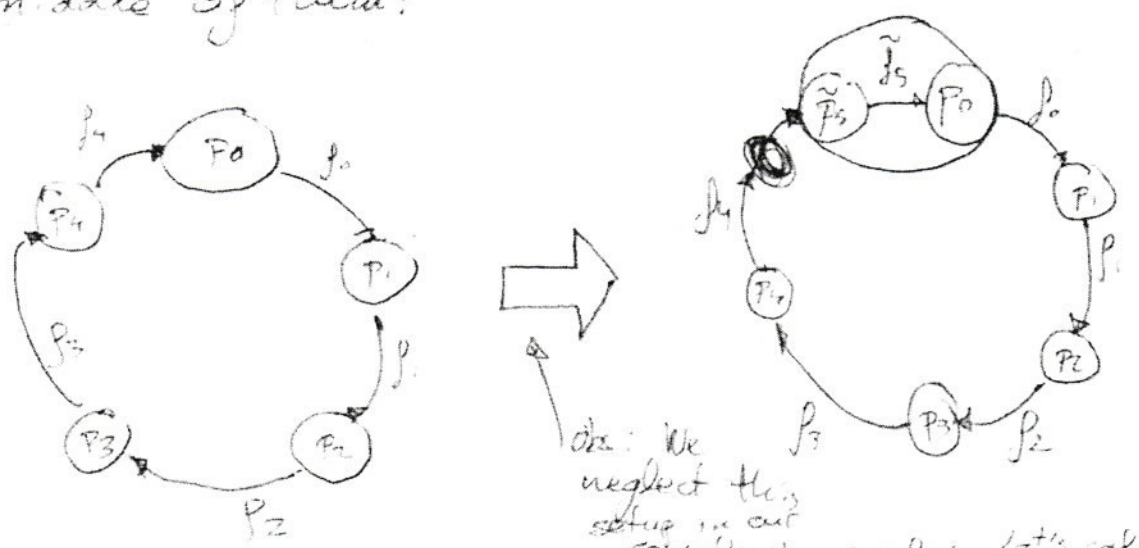
n.- p_{n-1} picks its right fork (f_{n-1}, which was free)

In this point we have reached a deadlock: Every philosopher p_i will try to pick $f_{i+1 \bmod n}$ which is now holded by $p_{i+1 \bmod n}$ who is waiting also for $f_{i+1 \bmod n}$. Thus no one will be able to get the left fork consequently not eating and not releasing the right fork → This deadlock leads also to starvation: No one will eat. So, the given solution does not work.

In the next page I will provide a solution based in graph coloring. And I will analyze its complexity.

Question 5 cont

To fix this problem I will propose a graph coloring based solution. But first let's ensure that we have a multiple of 2 number of forks by simulating in one of the philosophers, in case we do not have a multiple of 2 philosophers, the behavior of a pair of philosophers sitting together and the fork in the middle of them:

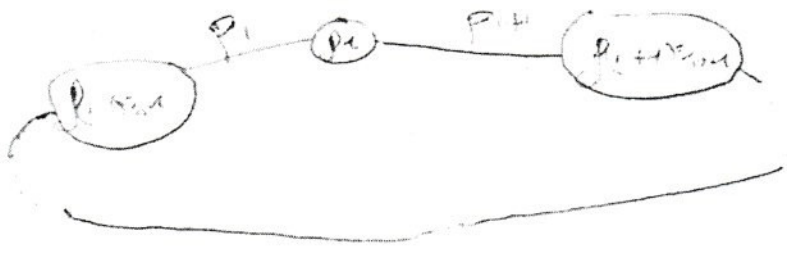


Ok: We neglect this setup in our complexity analysis. Let's call it phase 0 setup.

This will simplify the resources graph coloring.

"A ring graph with a multiple of two nodes has a coloring number of two where as a ring graph with a non multiple of two number of nodes has a coloring number of 3".

Now we have arranged the multiple of two problem, we are going to model the resource graph as follows



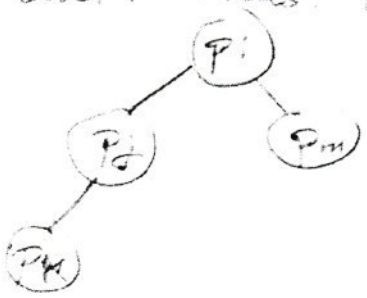
With our same P_i philosophers, f_i fork's notation.

QUESTION 5 cont.

We can now color the graph. We sort the nodes by degree but they all have degree one. We will pick one of them and assign color black. We run over all the circle assigning color black to all nodes whose neighbors are not colored and when we finish this round, we change color to white and set color white in a second loop to all remaining nodes. This initial computation is $O(n)$. As a result: Our forks are colored as in the figure below.

With this pre-setup we can define a very simple protocol: All philosophers seek the black fork first.

This will avoid the deadlock condition previously explained and also the starvation because a philosopher will only depend on one philosopher for one fork and two for the other. Thus the computing, as we can see in the tree is bounded $\checkmark 3R$ where



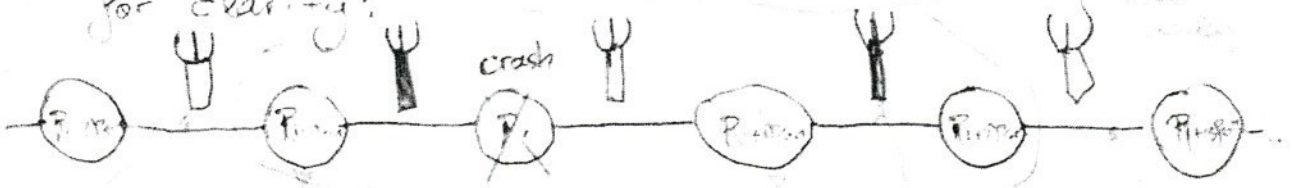
$R >> 1$ is the time taken for eating.

Next page, crashes considered.

Question 5 Cont

If a philosopher crashes executing the seek right fork first protocol while holding at least one fork, all other philosophers will starve: The processor in the right may eventually pick right fork and trying to get left it would never be able to thus blocking its right causing the same effect on the further right processor thus leading to a full starvation in the system.

If a philosopher crashes in my algorithm while holding both forks, the processor sharing the white fork with the crashed processor will not succeed in picking it thus blocking its black fork and starving and the processor sharing the black fork will not be able to block the black one thus ~~starving~~ starving but not blocking the white one. The processor sharing the ~~white~~ black one with the processor which was sharing white may also starve, but leaving its other neighbor free. So the total number of nodes starving is bounded to 3 apart from the crashing processor, which is a good result for the rest of the nodes in the system. See fig for clarity:

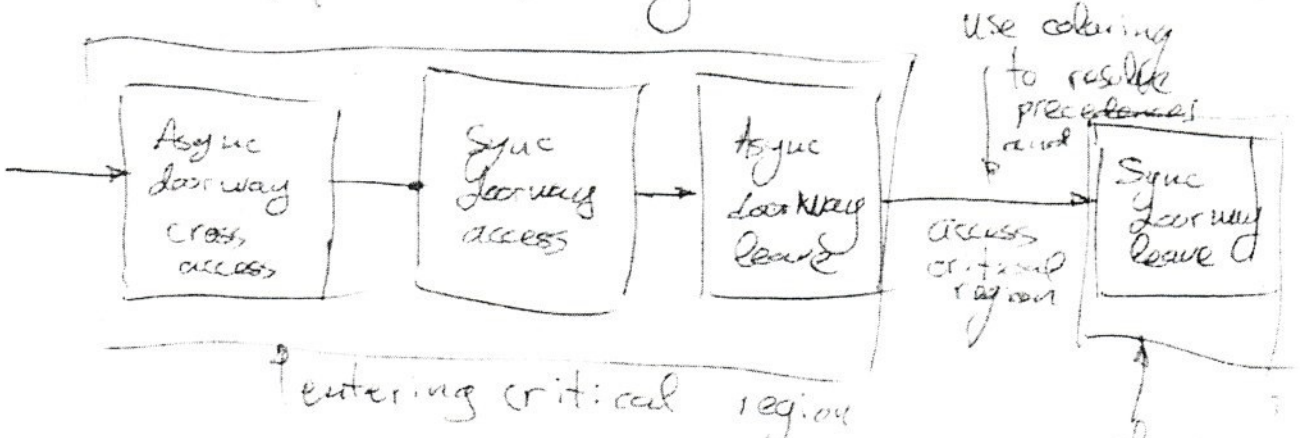


(6) QUESTION 6

Describe a polynomial, on the degree of the conflict (process) graph, solution for resource allocation problem.

Setup step: Coloring the ~~resource~~ conflict graph.
Time complexity: $O(n) * \#$ of diff colors $\sim$ degree (n = num nodes)
Comm complexity: $O(|E|)$ (|E| = num edges; correlated to)
This help us statically rank all the nodes to establish precedence levels to solve access conflicts.

Said so, we can use a doorways based protocol based on async and sync doorways crossing to isolate the critical region:



Each access and leave, either synch or asynch involves sending a multicast msg to all neighbors, so comm complexity is k times the degree of the power of the number of resources accessed $\rightarrow O(deg^{res})$ polynomial on the degree of conflict.