Midsem

1A:

Question: Does Acharya Badrinath algorithm work for FIFO channels?

Ans: No. Reason - any suitable example showing inconsistent snapshots.

Grading scheme - Correct ans + reason = 4

Partially correct reason - partial marks

No marks for only answer.

1B:

Answer: Forward proof (3 marks)

Considering consistency condition, prove why the max condition must hold. Proof by contradiction can be done ideally. Any other correct proof / correct explanation have been awarded marks as well.

Backward proof (3 marks)

Considering max condition, show that inconsistency is not possible. Again, proof by contradiction can be done. Thorough explanations have been awarded marks as well.

2A:

Why would you prefer the algorithm? (2 marks)
Reduction in Message Passing:

Since a process can enter the critical section if it has the token and is present anywhere in its request linked list (rather than only at the top), the number of token forwarding messages is reduced compared to the original queue-based version.
Instead of waiting for its turn and keep on forwarding token without actually executing CS foritself, in the modified algo, it executes the CS for itself, then

sends token, so the token does not have to come back again to that node for executing that node's request (if that node only has itself in the request list) . So, the return journey token exchanges are not required,thus reducing message passing.

Example:
Example to Illustrate the Reduction in Messages
Original Algorithm (Queue-Based) - More Token Transfers
Consider three processes in a tree:

P1 (Root)
P2 (Child of P1)
P3 (Child of P2)
The token is with P1, and the request queue is:
P3 → P2 → P1

Now, assume P1 and P3 both need to enter CS.

P3 requests CS → Sends request to P2 → P2 sends it to P1 (total 2 request messages).
P1 has the token, so it sends the token to P3 (total 2 token forwarding messages).
P3 enters CS and finishes execution.
The token must return back to P1 so that P1 can enter CS (total 2 more token messages).
Problem:
Even though P1 also needed CS, it had to give the token away first and then get it back, causing extra token movements.

Modified Algorithm (Linked List-Based) - Fewer Token Transfers
Same setup:

P1 has the token
Request linked list: P3 → P2 → P1
P1 and P3 both need CS
P3 requests CS → Sends request to P2, which forwards it to P1 (same 2 request messages).
P1 sees itself in the request linked list, so it enters CS immediately instead of forwarding the token away.
After finishing CS, P1 now sends the token directly to P3.
P3 finishes execution and forwards the token to the next process in the list (if

needed).
What is eliminated?
No need to send the token away and get it back (avoids the return journey token messages).
Token is forwarded only when necessary, reducing message passing.

Reasons Not to Prefer This Algorithm (2 marks)

Potential Starvation and Unfair Access: (1.5 marks)

The original queue-based approach ensures strict FIFO ordering, but the linked list version may allow some requests to be skipped if the current node request is present in the list when the token was passed, even if it requested earlier.
This could lead to certain processes experiencing starvation if their requests get delayed indefinitely.

Increased Local Overhead: (0.5 marks)

The process must now maintain a request linked list instead of a simple queue, leading to higher local processing overhead. To search for the current process in the linked list worst case time is O(n).

If any other valid reason provided for both advantages and disadvantages then upto 1 mark will be given for each reason.

2B:

Question:

What concerns do you see if the original Raymond's algorithm is run on directed graph that can have cycles instead of tree ?

Answer:

If thought revolves around what happens due to 2 paths diverging and tried to run the algorithm for that part - 2 Marks

And have mentioned any two of the below keywords (1/2+1/2=1)

i) Token Loss or Duplication

- In a tree, every node has a unique parent, ensuring a **clear request propagation path**.
- In a **cyclic directed graph**, a request might **loop indefinitely** or **duplicate itself**, leading to multiple token instances or loss of the only token.

ii). Deadlock Due to Cycles

- If the request forwarding follows cycles, there is a risk of **circular wait**, where processes keep waiting on each other in a loop without ever receiving the token.
- Example:
  - Process A requests the token from B.
  - B forwards it to C.
  - C forwards it back to A (forming a cycle).
  - Now, **none of them get the token**, causing a deadlock.

iii). Inefficient Token Routing

- In a tree, the token always moves **closer** to the requesting node.
- In a cyclic graph, the token may take a **longer or even infinite path** before reaching the intended node.

iv). Broken Request Queues

- The algorithm relies on **FIFO queues** to maintain request order.
- In a cycle, request queues may get **corrupted or inconsistent**, leading to **starvation** (a process never gets the token).

v). No Guaranteed Termination

- In a tree, a request propagates up until it finds the token holder.
- In a cyclic graph, a request may **keep bouncing indefinitely** between nodes, preventing termination.

2C:

Answer:

1. Lamport’s Mutual Exclusion Algorithm (Fully Permission-Based)

Message Complexity:

- **Synchronization Delay**: **O(N)**
  - After a site leaves the CS, the next process can only enter after receiving permission from all processes.

    N−1N-1
  - Since each process maintains a logical clock and timestamps requests, every process must check and acknowledge the next request in queue.
  - This results in messages.

    N−1N-1
- **Response Time**: **3(N-1)**
  - A request message is sent to **all N−1N-1 processes** → **(N-1) messages**.
  - Each process replies with a timestamped acknowledgment → **(N-1) messages**.
  - The process sends a release message after exiting CS → **(N-1) messages**.
  - **Total: 3(N-1) messages.**

2. Ricart-Agrawala Algorithm (Permission-Based)

Message Complexity:

- **Synchronization Delay**: **O(N)**
  - Similar to Lamport’s algorithm, the next process must wait for permission from all processes before entering CS.

    N−1N-1

  - Each process must check the timestamp and reply accordingly.
  - **Total delay = N−1N-1 responses, which is O(N).**
- **Response Time**: **2(N-1)**
  - A request message is sent to **all N−1N-1 processes** → **(N-1) messages**.
  - Each process replies if the requester has the highest priority → **(N-1) messages**.
  - Unlike Lamport, **no release messages** are required because mutual exclusion is ensured via direct request-reply.
  - **Total: 2(N-1) messages.**

---

3. Maekawa's Algorithm (Permission-Based with Quorum Selection)

Message Complexity:

- **Synchronization Delay**: **O(√N)**
  - Instead of requiring permission from all processes, it only needs permission from a **quorum of size O(N)O(\sqrt{N})**.

    N−1N-1

  - The next process needs to collect replies from its quorum set before entering CS.
  - **Total delay = O(N)O(\sqrt{N}).**
- **Response Time**: **3√N**
  - A request is sent to **√N quorum members** → **√N messages.**
  - Quorum members send **permission messages** → **√N messages**.
  - A **release message** is sent to the quorum after CS execution → **√N messages**.
  - **Total: 3√N messages.**

---

4. Suzuki-Kasami Algorithm (Token-Based)

Message Complexity:

- **Synchronization Delay**: **O(1)**
  - In a token-based approach, the process that exits CS simply passes the token to the next requesting process **immediately**.
  - Since token passing is direct, the next process doesn't have to wait for permission from multiple processes.
  - **Total delay = O(1) (just passing the token).**
- **Response Time**: **O(N)**
  - If a process **does not have the token**, it must send a request **to all N−1N-1 processes**.
  - The token holder then receives all requests and updates its queue.
  - When it releases CS, the token is passed to the next process based on the **request queue**.
  - **Total: O(N) messages.**

Why O(N) for Response Time?

- After releasing CS, the process must scan its request array (RN) to determine the next eligible process for token transfer.
- This takes **O(N) time** since it checks the entire list.

---

5. Raymond's Algorithm (Token-Based on Tree)

Message Complexity:

- **Synchronization Delay**: **O(log N)**
  - The token is **held in a logical tree structure**, meaning requests propagate **upward in O(log N) hops** before reaching the current token holder.
  - After a process exits CS, the token **travels back down the tree** to the next requester.

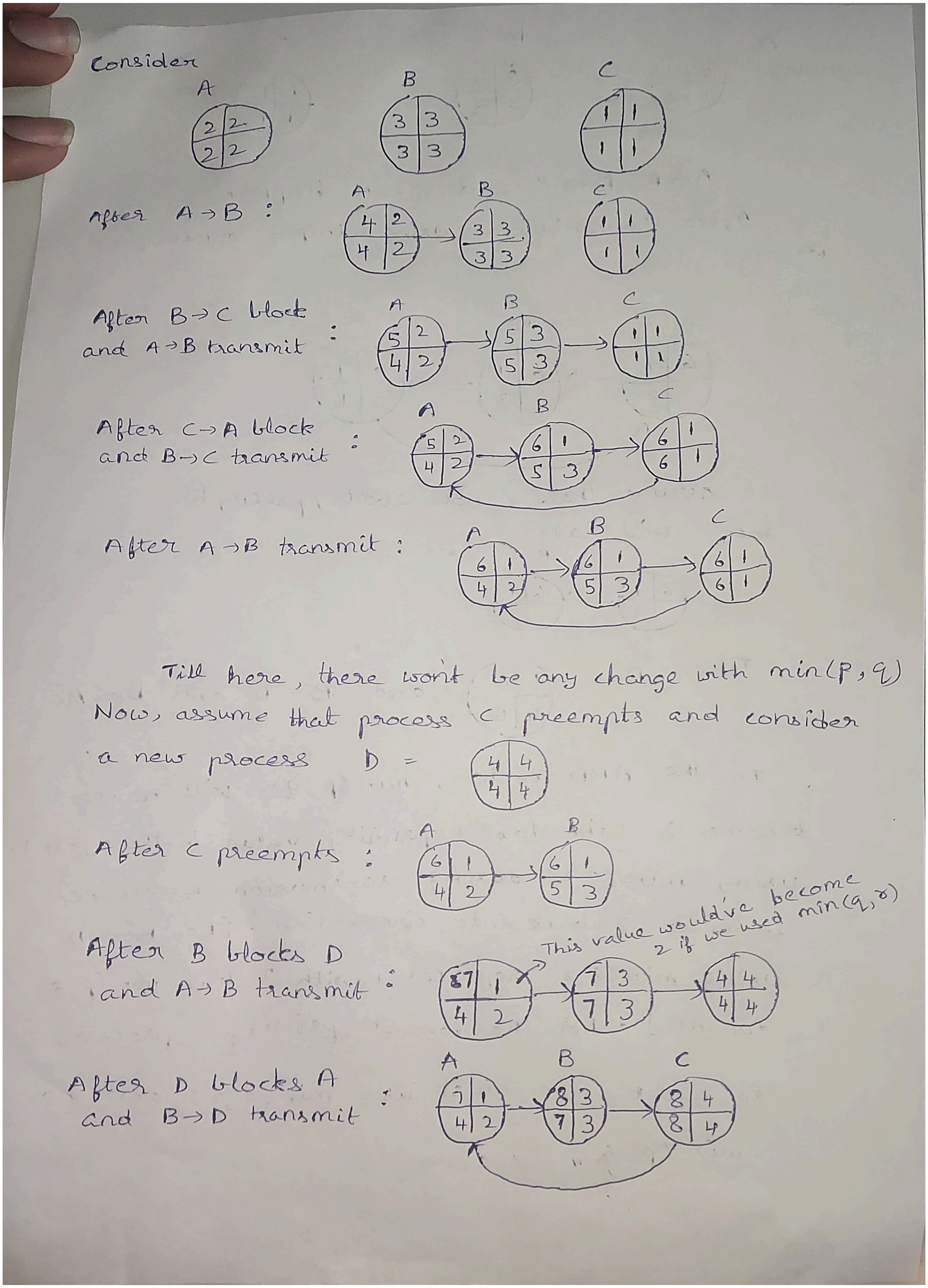
  - **Total delay = O(log N).**
- **Response Time**: **O(log N)**
  - If a process needs the token, it must send a request **up the tree** to the token holder.
  - The request is forwarded **along the tree structure**, taking **O(log N) hops**.
  - The token then moves down toward the requester.
  - **Total: O(log N) messages.**

Table: (Full Marks even with this or half just for writing the asymptotic time constraint like O(N) instead of O(3N-1))

Algorithm	**Sync Delay** (After CS Exit)	**Response Time** (From Request to CS Execution)
Lamport (Fully Permission-Based)	**O(N)**	**3(N-1)**
Ricart-Agrawala (Permission-Based)	**O(N)**	**2(N-1)**
Maekawa (Permission-Based, Quorum)	**O(√N)**	**3√N**
Suzuki-Kasami (Token-Based)	**O(1)**	**O(N)**
Raymond (Token-Based on Tree)	**O(log N)**	**O(log N)**

3A:

(i) The photo below illustrates a scenario in a priority-based algorithm for deadlock detection where the usage on min(p,q) instead of min(q,r) won't detect the deadlock. It shows the sequence of operations involving processes `A`, `D`, `C`, and `B`.



Consider

A
2 2
2 2
B
3 3
3 3
C
1 1
1 1

After A → B :

A
4 2
4 2
B
3 3
3 3
C
1 1
1 1

After B → C block and A → B transmit :

A
5 2
4 2
B
5 3
5 3
C
1 1
1 1

After C → A block and B → C transmit :

A
5 2
4 2
B
6 1
5 3
C
6 1
6 1

After A → B transmit :

A
6 1
4 2
B
6 1
5 3
C
6 1
6 1

Till here, there won't be any change with min(p, q)

Now, assume that process C preempts and consider a new process D =

4 4
4 4

After C preempts :

A
6 1
4 2
B
6 1
5 3

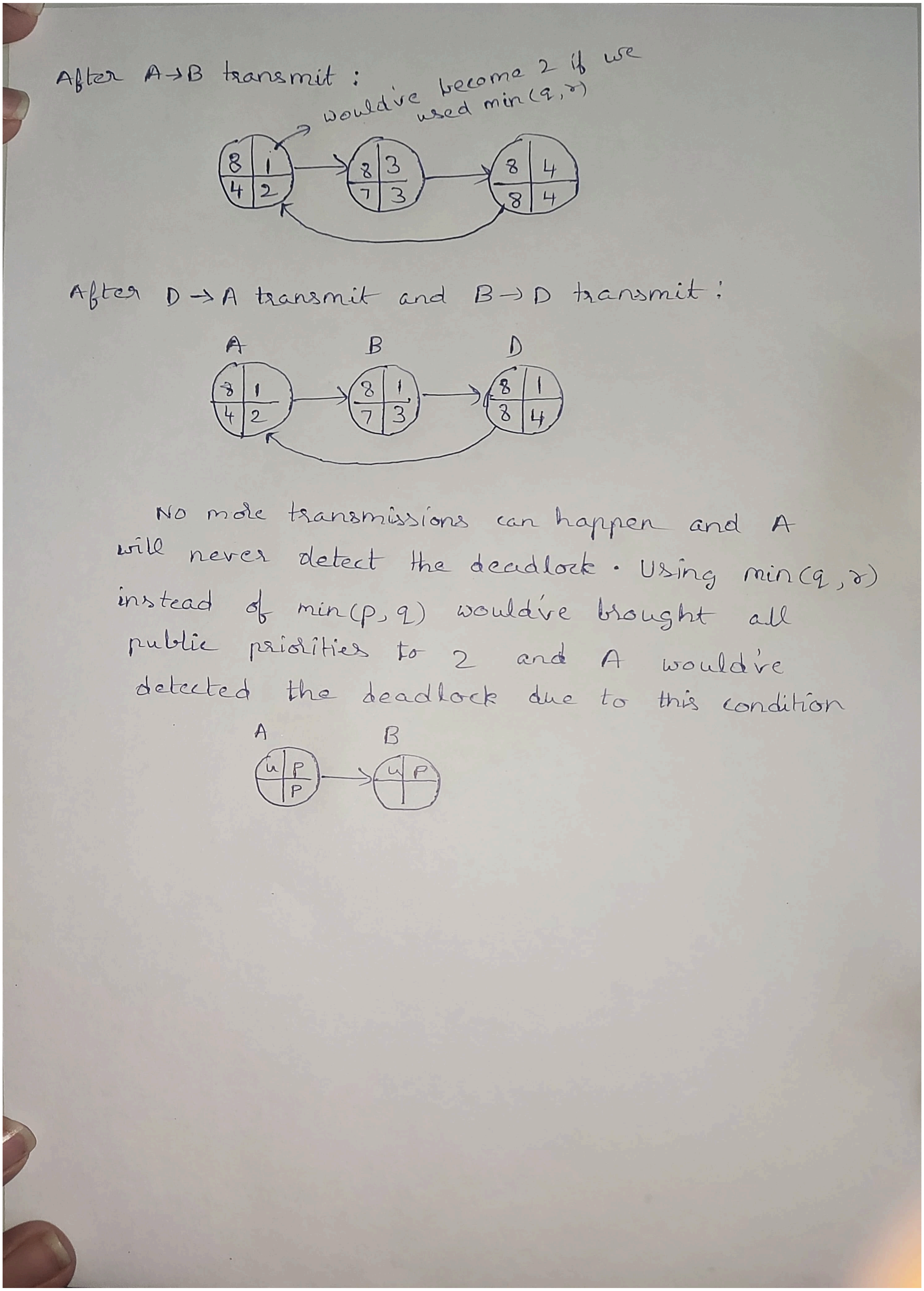
After B blocks D and A → B transmit :

87 1
4 2
7 3
7 3
4 4
4 4

This value would've become 2 if we used min(q, r)

After D blocks A and B → D transmit :

A
7 1
4 2
B
8 3
7 3
C
8 4
8 4



After A→B transmit :

would've become 2 if we used min(q, r)

8
1
4
2
8
3
7
3
8
4
8
4

After D→A transmit and B→D transmit :

A
B
D
8
1
4
2
8
1
7
3
8
1
8
4

No more transmissions can happen and A will never detect the deadlock. Using min(q, r) instead of min(p, q) would've brought all public priorities to 2 and A would've detected the deadlock due to this condition

A
B
u
p
p
u
p

Marking Scheme:

6 marks awarded for those who gave correct explanation or a correct counter-example.

3-4 marks awarded for those who came close the correct solution but missed some important steps.

Partial marks awarded in most cases based on the accuracy of reasoning.

4-6 marks will be awarded for any valid assumption for which the above counter example doesn't work. (You should write that there won't be any difference if we use min(p,q) or min(q,r))

3A:

(ii) Unique public numbers must be generated in the non-priority version, whereas they need not be unique in the priority version.

Priority Version:

In the priority version, if a deadlock occurs during the first transmission cycle, the highest public value is sent to all processes. Once all public values become equal, the lowest-priority process must detect the deadlock due to transmissions following the condition (u=v,p>q). Since the lowest priority process always detects the deadlock, uniqueness is not required.

Non-Priority Version:

Consider three processes:

Process 1 with a public value of 1

Process 2 with a public value of 2

Process 3 with a public value of 3

Assume that Process 1 blocks Process 2. If inc(1,2) = 4, the public value of Process 1 updates to 4.

Next, assume Process 2 blocks on Process 3. If inc(2,3) = 4 (which is not unique), the public value of Process 2 also becomes 4.

Now, Process 1 detects a deadlock because its public value matches that of Process 2 (both are 4), even though a real deadlock does not exist, demonstrating why unique public numbers are necessary in the non-priority version.

Marking Scheme:

A maximum of 0.5 marks were awarded if you just answer Yes/No without any explanation.

Full marks were awarded if you've provided correct explanation for atleast one of the versions. (You have to answer Yes/No for both the versions)

Maximum of 3 marks were awarded if you've arrived at a wrong verdict for any one of the versions. (Even though the explanation for the other part is correct).

Partial marks awarded in most cases based on the accuracy of reasoning.

Question:

Consider the Chandy-Misra-Haas probe-based algorithm for the AND model. Do you notice any concerns if multiple processes initiate the algorithm at the same time? How do you propose to fix the concern noticed?

Answer:

(3 marks)

If multiple processes initiate the Chandy-Misra-Haas algorithm simultaneously, each may detect a deadlock and **unnecessarily relinquish resources**, causing avoidable blocking and restarts. Unlike **Mitchell and Merritt's (0.5) algorithm**, which ensures only one process detects the deadlock.

This algorithm allows multiple initiators, leading to redundant terminations.

To fix this:

(2) — given 1 marks if written even something genuine.

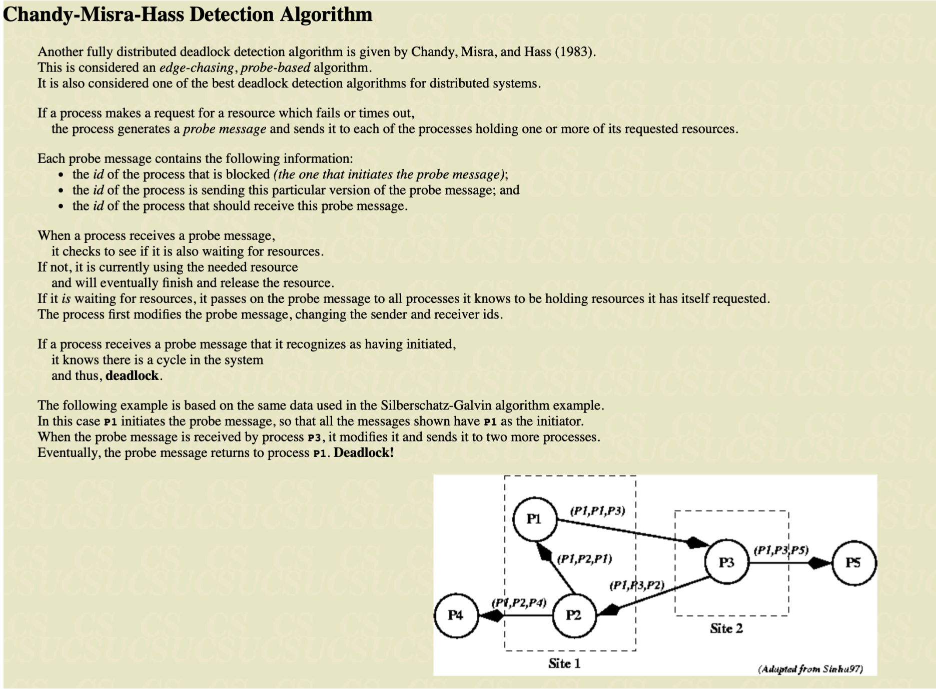
1. **Modify the algorithm to inform nodes in the cycle before aborting** – As a probe travels, it gathers nodes it has visited. The initiator can then notify these nodes before terminating, preventing unnecessary releases.

2. **Elect a single initiator** using timestamps or priority-based selection.

3. **Ignore duplicate detections** if another process has already started deadlock resolution.

4. **Use a centralized coordinator** to ensure only one detection cycle runs at a time.

These changes prevent redundant resource relinquishment, improving efficiency while keeping detection decentralized.



Chandy-Misra-Hass Detection Algorithm

Another fully distributed deadlock detection algorithm is given by Chandy, Misra, and Hass (1983).
This is considered an *edge-chasing*, *probe-based* algorithm.
It is also considered one of the best deadlock detection algorithms for distributed systems.

If a process makes a request for a resource which fails or times out, the process generates a *probe message* and sends it to each of the processes holding one or more of its requested resources.

Each probe message contains the following information:

- the *id* of the process that is blocked *(the one that initiates the probe message)*;
- the *id* of the process is sending this particular version of the probe message; and
- the *id* of the process that should receive this probe message.

When a process receives a probe message, it checks to see if it is also waiting for resources.
If not, it is currently using the needed resource and will eventually finish and release the resource.
If it *is* waiting for resources, it passes on the probe message to all processes it knows to be holding resources it has itself requested.
The process first modifies the probe message, changing the sender and receiver ids.

If a process receives a probe message that it recognizes as having initiated, it knows there is a cycle in the system and thus, **deadlock**.

The following example is based on the same data used in the Silberschatz-Galvin algorithm example.
In this case `P1` initiates the probe message, so that all the messages shown have `P1` as the initiator.
When the probe message is received by process `P3`, it modifies it and sends it to two more processes.
Eventually, the probe message returns to process `P1`. **Deadlock!**

(Adapted from Sinha97)