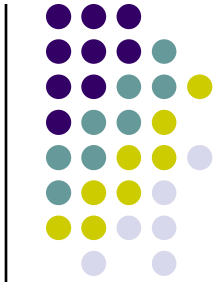
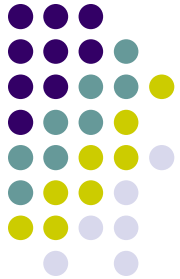


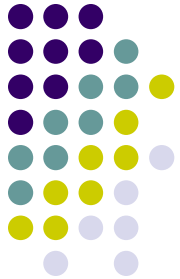
Distributed Mutual Exclusion

Mutual Exclusion



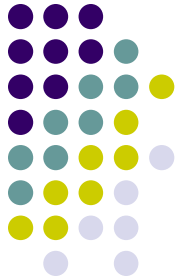
- Very well-understood in shared memory systems
- Requirements:
  - at most one process in critical section (safety)
  - if more than one requesting process, someone enters (liveness)
  - a requesting process enters within a finite time (no starvation)
  - requests are granted in some order (fairness)

Classification of Distributed Mutual Exclusion Algorithms



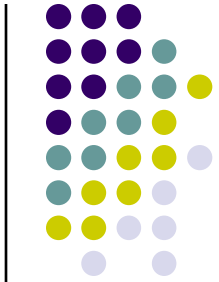
- Permission based
  - Node takes permission from all/subset of other nodes before entering critical section
  - Permission from all: costly, good for small systems
  - Permission from subset: scalable, widely used
    - Main problem: How to choose the subsets for each node?
- Token based
  - Single token in the system
  - Node enters critical section if it has the token
  - Algorithms differ in how the token is circulated among requesting nodes

Some Complexity Measures

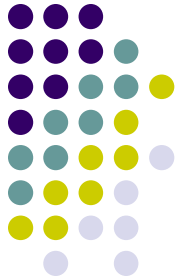


- No. of messages per critical section entry
- Synchronization delay
- Response time
- Throughput

Adaptive vs. Non-Adaptive

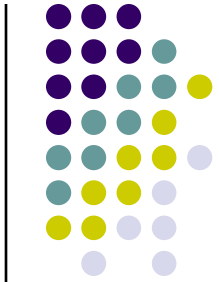


- Performance under low load (less requests for CS) should be better than performance under high load (lots of requests)
- Adaptive mutual exclusion algorithms: performance is dependent on load

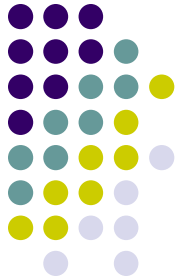


Permission based Algorithms

Lamport's Algorithm



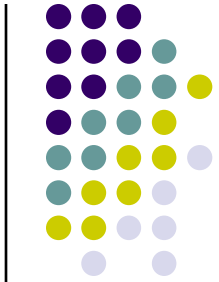
- Permission from all
- Every node i has a request queue q_i, keeps requests sorted by logical timestamps (total ordering enforced by including process id in the timestamps)
- To request critical section:
  - send timestamped REQUEST (ts_i, i) to all other nodes
  - put (ts_i, i) in its own queue
- On receiving a request (ts_i, i):
  - send timestamped REPLY to the requesting node i
  - put request (ts_i, i) in the queue



- To enter critical section:
  - i enters critical section if (ts_i, i) is at the top if its own queue, and i has received a message (any message) with timestamp larger than (ts_i, i) from ALL other nodes.

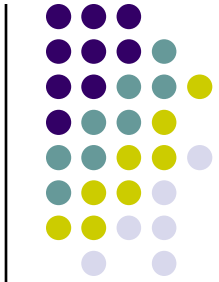
- To release critical section:
  - i removes its request from its own queue and sends a timestamped RELEASE message to all other nodes
  - On receiving a RELEASE message from i, i's request is removed from the local request queue

Some points to note

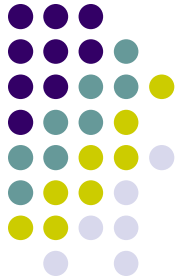


- Purpose of REPLY messages from node i to j is to ensure that j knows of all requests of i prior to sending the REPLY (and therefore, possibly any request of i with timestamp lower than j's request)
- Requires FIFO channels.
- 3(n – 1) messages per critical section invocation
- Synchronization delay = max. message transmission time
- Requests are granted in order of increasing timestamps

Ricart-Agarwala Algorithm

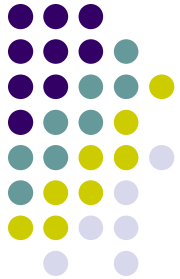


- Improvement over Lamport's
- Main Idea:
  - node j need not send a REPLY to node i if j has a request with timestamp lower than the request of i (since i cannot enter before j anyway in this case)
- Does not require FIFO
- 2(n – 1) messages per critical section invocation
- Synchronization delay = max. message transmission time
- requests granted in order of increasing timestamps

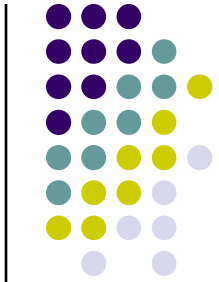


- To request critical section:
  - send timestamped REQUEST message (ts_i, i)
- On receiving request (ts_i, i) at j:
  - send REPLY to i if j is neither requesting nor executing critical section or if j is requesting and i's request timestamp is smaller than j's request timestamp. Otherwise, defer the request.
- To enter critical section:
  - i enters critical section on receiving REPLY from all nodes
- To release critical section:
  - send REPLY to all deferred requests

Maekawa's Algorithm

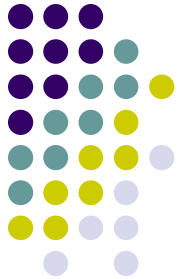


- Permission obtained from only a subset of other processes, called the Request Set (or Quorum)
- Separate Request Set R_i for each process i
- Requirements:
  - for all i, j: $R_i \cap R_j \neq \Phi$
  - for all i: $i \in R_i$
  - for all i: $|R_i| = K$, for some K
  - any node i is contained in exactly D Request Sets, for some D
- K = D (easy to see)
- For minimum K, $K \approx \sqrt{N}$ (why?)

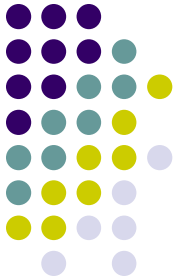


A simple version

- To request critical section:
  - i sends REQUEST message to all process in R_i
- On receiving a REQUEST message:
  - send a REPLY message if no REPLY message has been sent since the last RELEASE message is received. Update status to indicate that a REPLY has been sent. Otherwise, queue up the REQUEST
- To enter critical section:
  - i enters critical section after receiving REPLY from all nodes in R_i



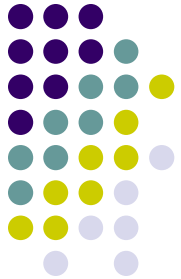
- To release critical section:
  - send RELEASE message to all nodes in R_i
  - On receiving a RELEASE message, send REPLY to next node in queue and delete the node from the queue. If queue is empty, update status to indicate no REPLY message has been sent since last RELEASE is received.



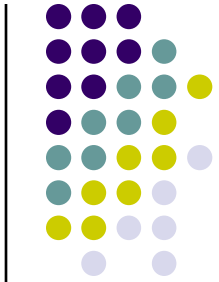
- Message Complexity: 3*sqrt(N)
- Synchronization delay =

  2 *(max message transmission time)
- Major problem: Deadlock possible
- Can you update the protocol with additional messages to solve this problem?
  - Good practice ☺
  - Maekawa's protocol already does that, we just looked at a part of it
- Building the request sets?

Some Points

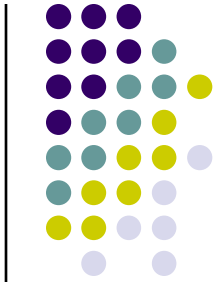


- Permission based algorithms with permission from a subset are widely used
  - Voting/Quorum based protocols
    - In Maekawa's algorithm,
      - each process has one vote
      - A process needs a certain number of votes to proceed
- Questions/Issues
  - How to choose the quorums?
  - Should the quorum be the same for read and write?
  - Should each process have one vote only? Same number of votes for all?
  - Dynamic quorums/votes



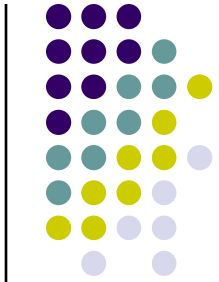
Token based Algorithms

Token based Algorithms



- Single token circulates, enter CS when token is present
- Mutual exclusion obvious
- Algorithms differ in how to find and get the token
  - Token circulates, nodes use it when it passes through them
  - Token stays at node of last use, other nodes request for it when needed
    - Need to differentiate between old and current requests

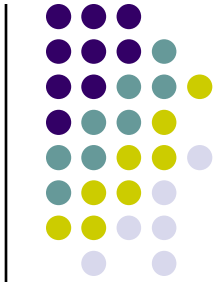
Suzuki Kasami Algorithm



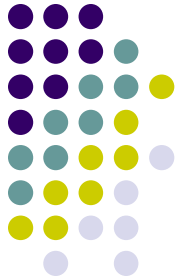
- Broadcast a request for the token
- Process with the token sends it to the requestor if it does not need it

Issues:

- Current vs. outdated requests
- Determining sites with pending requests
- Deciding which site to give the token to

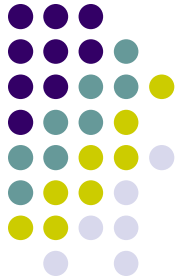


- The token:
  - Queue (FIFO) Q of requesting processes
  - LN[1..n] : sequence number of request that j executed most recently
- The request message:
  - REQUEST(i, k): request message from node i for its k^{th} critical section execution
- Other data structures
  - RN_i[1..n] for each node i, where RN_i[j] is the largest sequence number received so far by i in a REQUEST message from j.



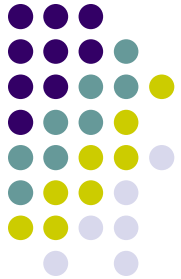
- To request critical section:
  - If i does not have token, increment $RN_i[i]$ and send REQUEST(i, $RN_i[i]$) to all nodes
  - if i has token already, enter critical section if the token is idle (no pending requests), else follow rule to release critical section

- On receiving REQUEST(i, sn) at j:
  - set $RN_j[i] = \max(RN_j[i], sn)$
  - if j has the token and the token is idle, send it to i if $RN_j[i] = LN[i] + 1$. If token is not idle, follow rule to release critical section



- To enter critical section:
  - enter CS if token is present

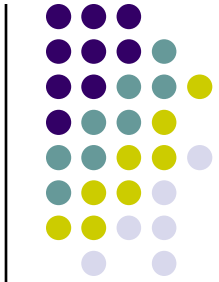
- To release critical section:
  - set LN[i] = $RN_i[i]$
  - For every node j which is not in Q (in token), add node j to Q if $RN_i[j] = LN[j] + 1$
  - If Q is non empty after the above, delete first node from Q and send the token to that node



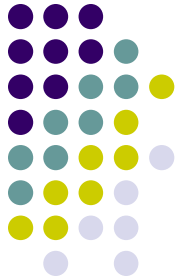
Points to note:

- No. of messages: 0 if node holds the token already, n otherwise
- Synchronization delay: 0 (node has the token) or max. message delay (token is elsewhere)
- No starvation

Raymond's Algorithm

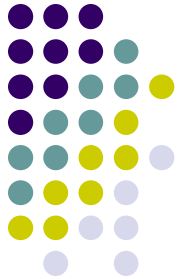


- Forms a directed tree (logical) with the token-holder as root
- Each node has variable "Holder" that points to its parent on the path to the root. Root's Holder variable points to itself
- Each node i has a FIFO request queue Q_i



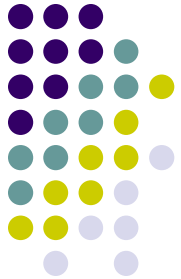
- To request critical section:
  - Send REQUEST to parent on the tree, provided i does not hold the token currently and Q_i is empty. Then place request in Q_i

- When a non-root node j receives a request from i
  - place request in Q_j
  - send REQUEST to parent if no previous REQUEST sent



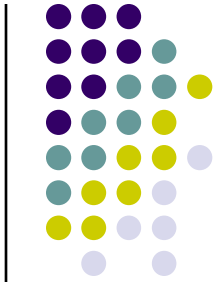
- When the root r receives a REQUEST
  - place request in Q_r
  - if token is idle, follow rule for releasing critical section (shown later)

- When a node receives the token
  - delete first entry from the queue
  - send token to that node (maybe itself)
  - set Holder variable to point to that node
  - if queue is non-empty, send a REQUEST message to the parent (node pointed at by Holder variable)



- To execute critical section
  - enter if token is received and own entry is at the top of the queue; delete the entry from the queue

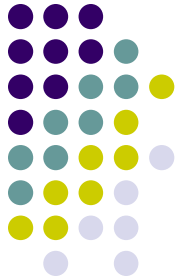
- To release critical section
  - if queue is non-empty, delete first entry from the queue, send token to that node and make Holder variable point to that node
  - If queue is still non-empty, send a REQUEST message to the parent (node pointed at by Holder variable)



Points to note:

- Avg. message complexity O(log n)
- Sync. delay (T log n)/2, where T = max. message delay

L - Exclusion



- At most L processes can be in critical section at any one time
- Can be implemented with L tokens
- Other algorithms exist