

Mutual Exclusion in a Distributed Environment (Review)

- Mutual exclusion
 - Centralized algorithms
 - Central physical clock
 - Central coordinator
 - Distributed algorithms
 - Time-based event ordering
 - Lamport's algorithm (logical clocks)
 - Ricart & Agrawala's algorithm (" ")
 - Suzuki & Kasimi's algorithm (broadcast)
 - Token passing
 - Le Lann's token-ring algorithm (logical ring)
 - Raymond's tree algorithm (logical tree)
 - Sharing K identical resources
 - Raymond's extension to Ricart & Agrawala's time-based algorithm
 - Atomic transactions (later in course)
- Related — self-stabilizing algorithms, election, agreement, deadlock

1

Fall 2005, Lecture 09

Suzuki and Kasami's Broadcast Algorithm (1985)

- Overview:
 - If a thread wants to enter the critical section, and it does not have the token, it broadcasts a *request* message to all other sites in the token's request set
 - The thread that has the token will then send it to the requesting thread
 - However, if it's in the critical section, it gets to finish before sending the token
 - A thread holding the token can continuously enter the critical section until the token is requested
 - Request vector at thread i :
 - $RN_i[k]$ contains the largest sequence number received from thread k in a *request* message
 - Token consists of vector and a queue:
 - $LN[k]$ contains the sequence number of the latest executed request from thread k
 - Q is the queue of requesting threads

2

Fall 2005, Lecture 09

Suzuki and Kasami's Broadcast Algorithm (cont.)

- Requesting the critical section (CS):
 - When a thread i wants to enter the CS, if it does not have the token, it:
 - Increments its sequence number sn and its request vector $RN_i[j]$ to $RN_i[j]+1$
 - Sends a *request* message containing new sn to all threads in that CS's request set
 - When a thread k receives the *request* message, it:
 - Sets $RN_k[j]$ to $\text{MAX}(RN_k[j], sn \text{ received})$
 - If $sn < RN_k[j]$, the message is outdated
 - If thread k has the token and is not in the CS (i.e., is not using it), and if $RN_k[j] == LN[j]+1$ (indicating an outstanding request) it sends the token to thread i
- Executing the CS:
 - A thread enters the CS when it has acquired the token

3

Fall 2005, Lecture 09

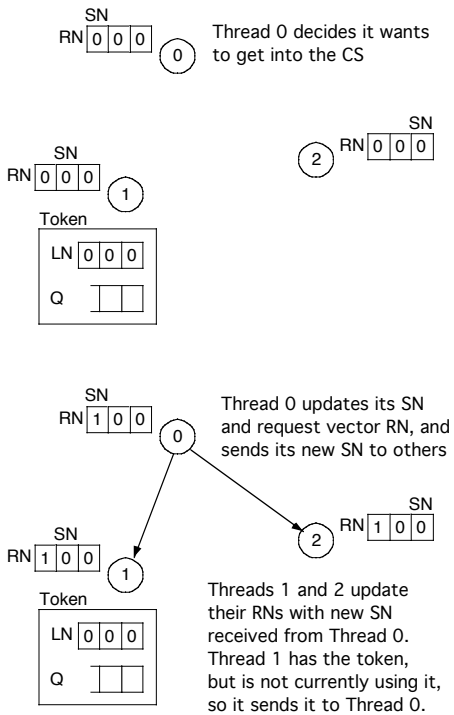
Suzuki and Kasami's Broadcast Algorithm (cont.)

- Releasing the CS:
 - When a thread i leaves the CS, it:
 - Sets $LN[j]$ of the token equal to $RN_i[j]$
 - Indicates that its request $RN_i[j]$ has been executed
 - For every thread k whose ID is not in the token queue Q , it appends its ID to Q if $RN_i[k] == LN[k]+1$
 - Indicates that thread k has an outstanding request
 - If the token queue Q is nonempty after this update, it deletes the thread ID at the head of Q and sends the token to that thread
 - Gives priority to others' requests
 - Otherwise, it keeps the token
- Evaluation:
 - 0 to N messages required to enter CS
 - No messages if thread holds the token
 - Otherwise $N-1$ requests, 1 reply

4

Fall 2005, Lecture 09

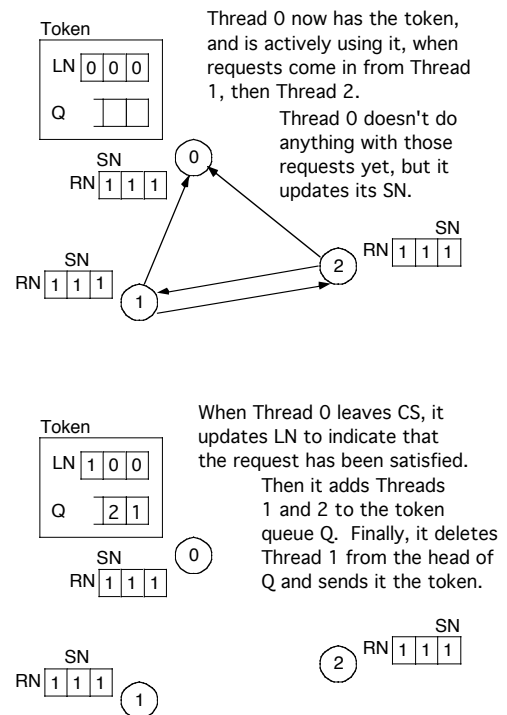
Suzuki and Kasami's Broadcast Algorithm (cont.)



5

Fall 2005, Lecture 09

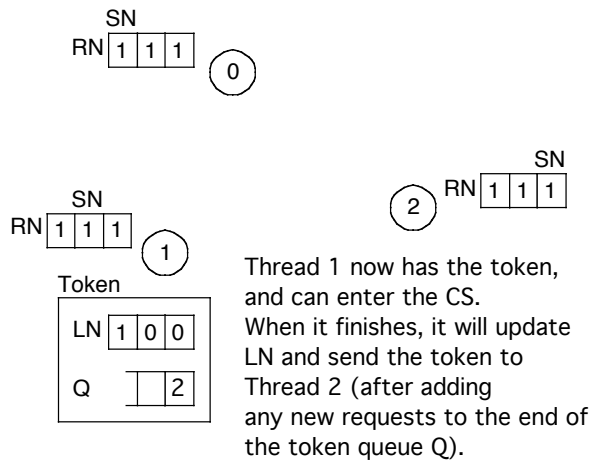
Suzuki and Kasami's Broadcast Algorithm (cont.)



6

Fall 2005, Lecture 09

Suzuki and Kasami's Broadcast Algorithm (cont.)



7

Fall 2005, Lecture 09

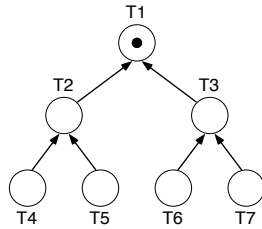
Token-Ring Algorithm (Le Lann, 1977 ?)

- Processes are arranged in a logical ring
- At start, process 0 is given a *token*
 - Token circulates around the ring in a fixed direction via point-to-point messages
 - When a process acquires the token, it has the right to enter the critical section
 - After exiting CS, it passes the token on
- Evaluation:
 - N-1 messages required to enter CS
 - Not difficult to add new processes to ring
 - With unidirectional ring, mutual exclusion is fair, and no process starves
 - ✗ Not very fault-tolerant
 - ✗ Difficult to detect when token is lost
 - ✗ Doesn't guarantee "happened-before" order of entry into critical section

8

Fall 2005, Lecture 09

Raymond's Tree Algorithm (1989)



Overview:

- Threads are arranged as a *logical tree*
 - Edges are directed toward the thread that holds the token (called the “holder”, which also acts as the root of logical tree)
- Each thread has:
 - A variable *holder* that points to its immediate neighbor on the directed path toward the holder of the token
 - A FIFO queue called *request_q* that holds its requests for the token, as well as any requests from neighbors that have requested but haven't received the token
 - If *request_q* is non-empty, that implies the node has already sent the request at the head of its queue toward the holder

9

Fall 2005, Lecture 09

Raymond's Tree Algorithm (cont.)

■ Requesting the critical section (CS):

- When a thread wants to enter the CS, but it does not have the token, it:
 - Adds its request to its *request_q*
 - If its *request_q* was empty before the addition, it sends a *request* message along the directed path toward the holder
 - If the *request_q* was not empty, it's already made a request, and has to wait
- When a thread in the path between the requesting thread and the holder receives the *request* message, it
 - < same as above >
- When the holder receives a *request* message, it
 - Sends the token (in a message) toward the requesting thread
 - Sets its *holder* variable to point toward that thread (toward the new holder)

10

Fall 2005, Lecture 09

Raymond's Tree Algorithm (cont.)

■ Requesting the CS (cont.):

- When a thread in the path between the holder and the requesting thread receives the token, it
 - Deletes the top entry (the most current requesting thread) from its *request_q*
 - Sends the token toward the thread referenced by the deleted entry, and sets its *holder* variable to point toward that thread
 - If its *request_q* is not empty after this deletion, it sends a *request* message along the directed path toward the new holder (pointed to by the updated *holder* variable)

■ Executing the CS:

- A thread can enter the CS when it receives the token **and** its own entry is at the top of its *request_q*
 - It deletes the top entry from the *request_q*, and enters the CS

11

Fall 2005, Lecture 09

Raymond's Tree Algorithm (cont.)

■ Releasing the CS:

- When a thread leaves the CS
 - If its *request_q* is not empty (meaning a thread has requested the token from it), it:
 - Deletes the top entry from its *request_q*
 - Sends the token toward the thread referenced by the deleted entry, and sets its *holder* variable to point toward that thread
 - If its *request_q* is not empty after this deletion (meaning more than one thread has requested the token from it), it sends a *request* message along the directed path toward the new holder (pointed to by the updated *holder* variable)

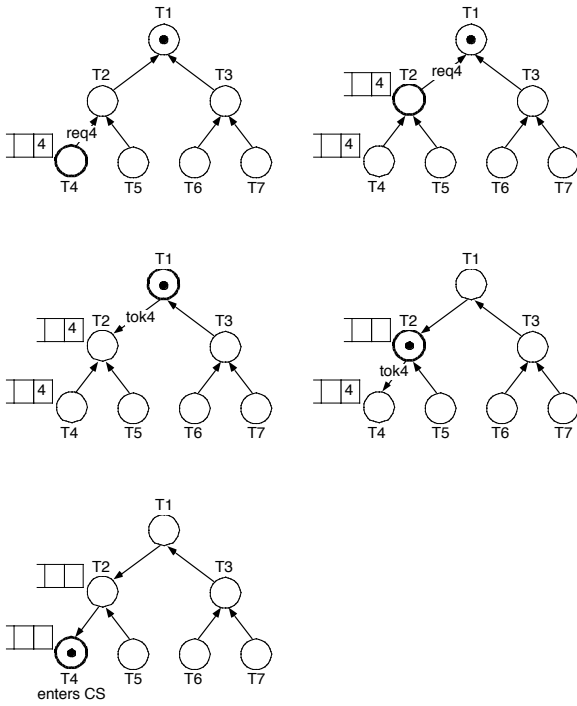
■ Evaluation:

- ✓ On average, $O(\log N)$ messages required to enter CS
 - Average distance between any two nodes in a tree with N nodes is $O(\log N)$

12

Fall 2005, Lecture 09

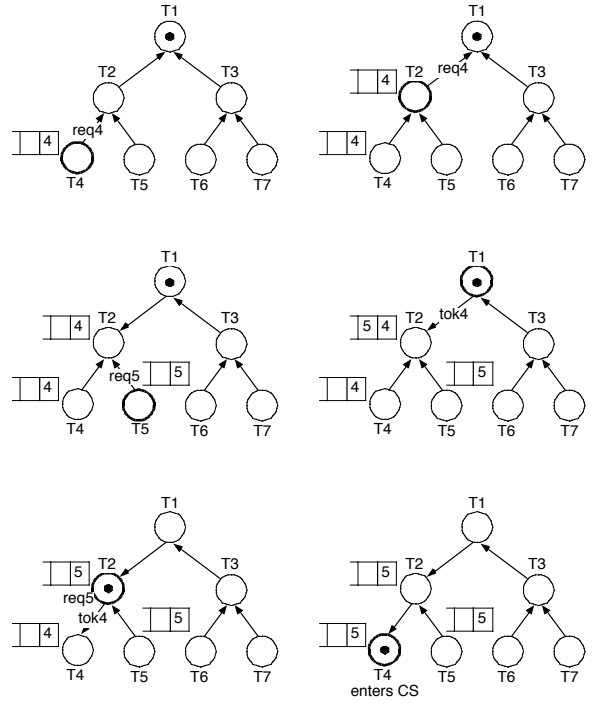
Raymond's Tree Algorithm (cont.)



13

Fall 2005, Lecture 09

Raymond's Tree Algorithm (cont.)



14

Fall 2005, Lecture 09