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

Ricart and Agrawala's Algorithm (1981)

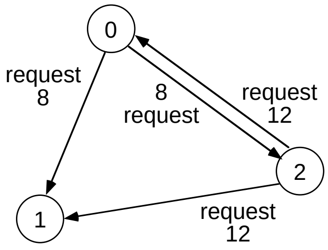
- Requesting the critical section (CS):
  - When a thread wants to enter the CS, it:
    - Sends a timestamped *request* message to all threads in that CS's request set
  - When a thread receives a *request* message:
    - If it is neither requesting nor executing the CS, it returns a *reply* message
    - If it is requesting the CS, but the timestamp on the incoming request is smaller than the timestamp on its own request, it returns a *reply* message
      - Means the other thread requested first
    - Otherwise, it defers answering the request
- Executing the CS:
  - A thread enters the CS when:
    - It has received a *reply* message from all other threads in the request set

Ricart and Agrawala's Algorithm (cont.)

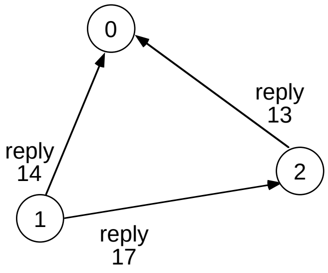
- Releasing the CS:
  - When a thread leaves the CS, it:
    - Sends a *reply* message to all the deferred requests
    - (Thread with next earliest request will now received its last *reply* message and enter the CS)
- Evaluation:
  - 2(N–1) messages required to enter CS
    - (N–1) reply, (N–1) request
- Evaluation (Lamport, Ricart & Agawala):
  - ✘ Distributed performance bottleneck
  - ✘ Now N points of failure
    - If a thread crashes, it fails to reply, which is interpreted as a denial of permission to enter the CS, so everyone waits…
  - ✘ Need up-to-date group communication

Ricart and Agrawala's Algorithm (cont.)

- Both threads 0 and 2 request the CS:



- Threads 1 and 2 reply, thread 0 defers and enters the CS since its request was first:



- After leaving the CS, thread 0 replies to thread 2, which enters the CS

Raymond's Extension For Sharing K Identical Resources (1987)

- K identical resources, which must be shared among N processes
- Raymond's extension to Ricart and Agrawala's algorithm:
  - A process can enter the CS as soon as it has received N–K *reply* messages
  - Algorithm is generally the same as R&A, with one difference:
    - R&A — *reply* messages arrive only when process is waiting to enter CS
    - Raymond —
      - N–K *reply* messages arrive when process is waiting to enter CS
      - Remaining K–1 *reply* messages can arrive when process is in the CS, after it leaves the CS, or when it's waiting to enter the CS again
      - Must keep a count of number of outstanding *reply* messages, and not count those toward next set of replies

---

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 thread

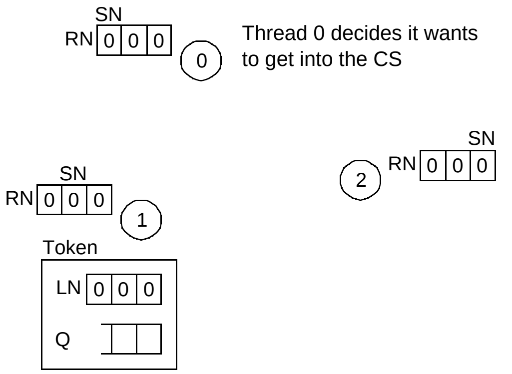
Suzuki and Kasami's Broadcast Algorithm (cont.)

- 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 $RN_i[i]$ to $RN_i[i]+1$
    - 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 $RN_k[i]$ to MAX($RN_k[i]$, *sn* received)
      - If $sn < RN_k[i]$, the message is outdated
    - If thread *k* has the token and is not in the CS (i.e., is not using it), and if $RN_k[i] == LN[i]+1$ (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

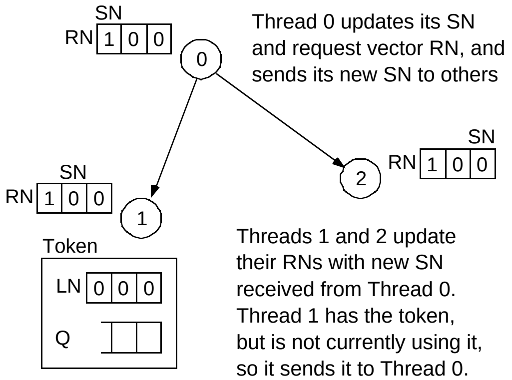
Suzuki and Kasami's Broadcast Algorithm (cont.)

- Releasing the CS:
  - When a thread *i* leaves the CS, it:
    - Sets LN[*i*] of the token equal to $RN_i[i]$
      - Indicates that its request $RN_i[i]$ 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

Suzuki and Kasami's Broadcast Algorithm (cont.)



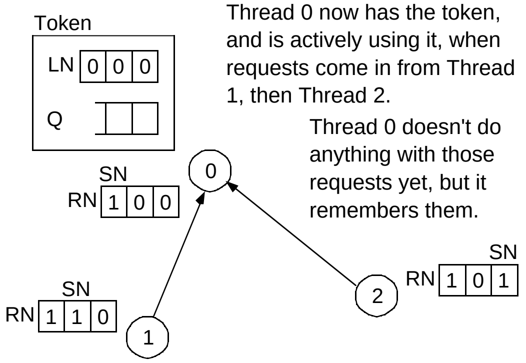
Thread 0 decides it wants to get into the CS



Thread 0 updates its SN and request vector RN, and sends its new SN to others

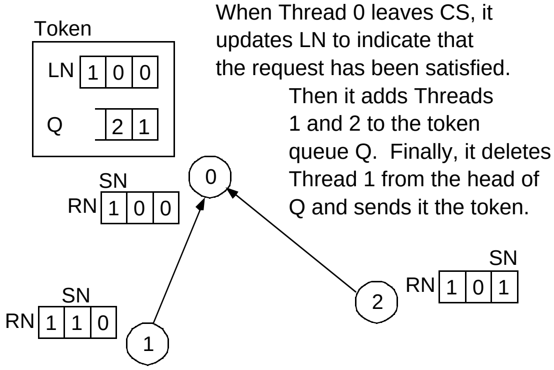
Threads 1 and 2 update their RNs with new SN received from Thread 0. Thread 1 has the token, but is not currently using it, so it sends it to Thread 0.

Suzuki and Kasami's Broadcast Algorithm (cont.)



Thread 0 now has the token, and is actively using it, when requests come in from Thread 1, then Thread 2.

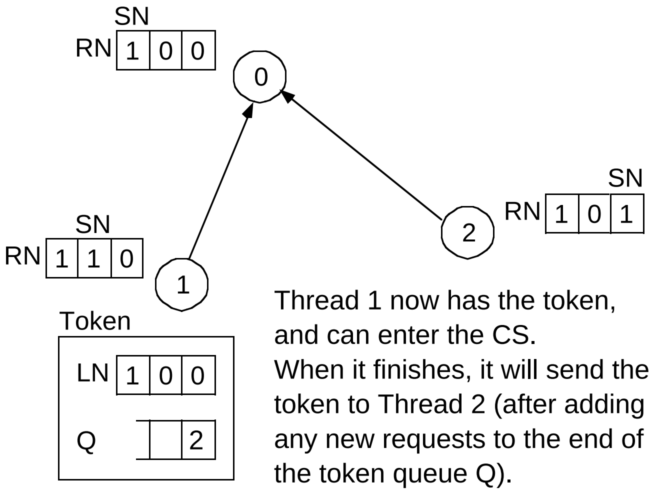
Thread 0 doesn't do anything with those requests yet, but it remembers them.



When Thread 0 leaves CS, it updates LN to indicate that the request has been satisfied.

Then it adds Threads 1 and 2 to the token queue Q. Finally, it deletes Thread 1 from the head of Q and sends it the token.

Suzuki and Kasami's Broadcast Algorithm (cont.)

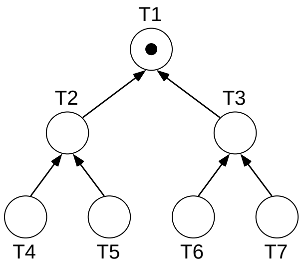


Thread 1 now has the token, and can enter the CS. When it finishes, it will send the token to Thread 2 (after adding any new requests to the end of the token queue Q).

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

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", initially the root of tree)
  - Each thread has:
    - A variable *holder* that points to its 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

---

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)

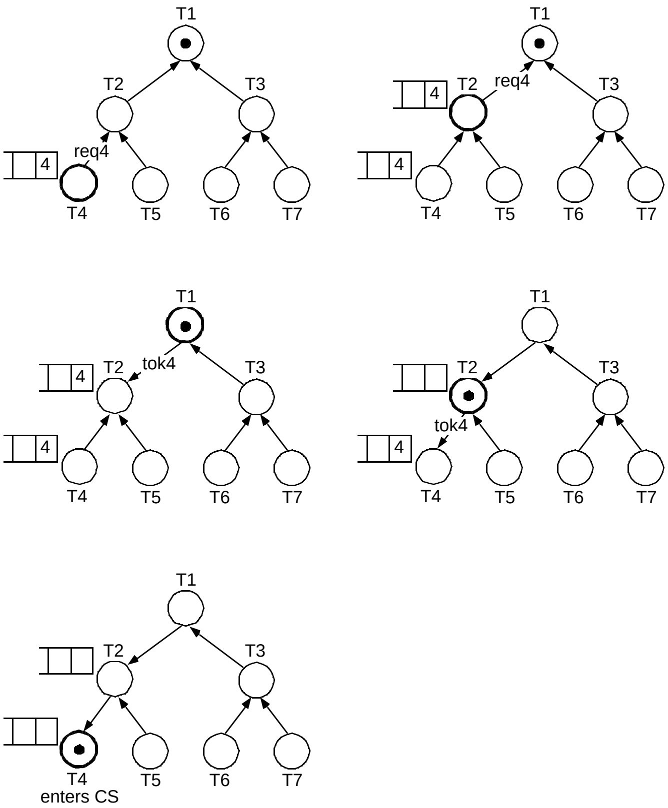
Raymond's Tree Algorithm (cont.)

- 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

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*)

Raymond's Tree Algorithm (cont.)



Raymond's Tree Algorithm (cont.)