Synchronization / Mutual Exclusion in a Centralized Environment

- User programs threads to explicitly coordinate with each other
  - Dijkstra’s Algorithm 1, 2, and 3
  - Dekker’s Algorithm, Peterson’s Algorithm
- OS provides support
  - Semaphores
  - Locks and condition variables
  - Monitors
  - Critical regions, path expressions, etc.
- Architectural support can make implementation easier
  - Interrupts
  - Atomic read-modify-write instructions
    - Test-and-set
    - Swap

Dijkstra’s Algorithms for Cooperating Processes (1965)

Algorithm 1 (Both Processes Shown)

```
process1 ( ) {
  while (1) {
    while (turn != 1)
      ; /* do nothing */
    …critical section…
    turn = 2;
    …non-critical code…
  }
}
```

```
process2 ( ) {
  while (1) {
    while (turn != 2)
      ; /* do nothing */
    … critical section…
    turn = 1;
    …non-critical code…
  }
}
```

Algorithm 2- (1 Process)

```
process1 ( ) {
  while (1) {
    while (p2InCS)
      ; /* do nothing */
    p1InCS = true;
    …critical section…
    p1InCS = false;
    …non-critical code…
  }
}
```

Algorithm 2 (1 Process)

```
process1 ( ) {
  while (1) {
    p1InCS = true;
    while (p2InCS)
      ; /* nothing */
    … critical section…
    p1InCS = false;
    …non-critical …
  }
}
```

Peterson’s Algorithm (1981)

```
process1 ( ) {
  while (1) {
    interested[1] = true;
    turn = 2;
    while (interested[2] && turn==2)
      ; /* do nothing */
    …critical section…
    interested[1] = false;
    …non-critical code…
  }
}
```

- Operation:
  - *interested*[*i*]==true indicates process *i* is interested in getting into the critical section
  - *turn* is used to break ties
    - Each insists it’s the other’s turn
    - Since memory write is atomic, even if both processes are almost in lock-step, one will succeed in insisting the other go first

---

Lamport’s Bakery Algorithm (For *n* Processes) (1974)

```
process-i ( ) {
  while (1) {
    choosing_num[i] = true;
    num[i] =
      max(num[0], num[1], … , num[n–1]) + 1;
    choosing_num[i] = false;

    for ( k=0 ; k < n–1 ; k++) {
      while (choosing[k])
        ; /* do nothing */
      while (  (num[k] != 0)  &&
        ( (num[k] < num[i]) ||
          (num[k] == num[i] && k < i) )  )
        ; /* do nothing */

    …critical section…

    num[i] = 0;

    …non-critical code…
  }
}
```

Mutual Exclusion in a Distributed Environment

- 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

Mutual Exclusion in a Distributed Environment — General Requirements

- N processes share a single resource, and require mutually-exclusive access
- Conditions to satisfy:
  - A process holding the resource must release it before it can be granted to another process
  - Requests for the resource must be granted in the order in which they're made
  - If every process granted the resource eventually releases it, then every request will be eventually granted
- Assumptions made:
  - Messages between two processes are received in the order they are sent
  - Every message is eventually received
  - Each process can send a message to any other process

Central Physical Clock

- Provide a single central physical clock, just like in a centralized system
  - Processes request physical timestamps from this clock and use them to order events

✔ Advantages:

- Simplicity

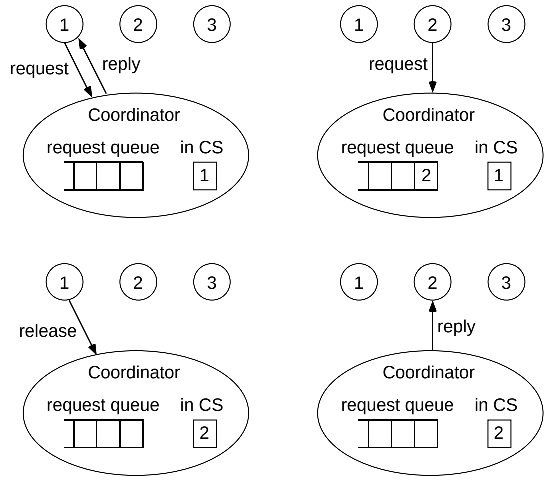
✗ Disadvantages:

- Clock must always be available to provide the requested timestamps
- Transmission errors can prevent the proper ordering from taking place
- An accurate estimation of transmission delays is required
- The degree of accuracy may not be as high as desired

Central Coordinator

- To enter the critical section, a thread sends a *request* message to the central coordinator, and waits for a reply
- When the coordinator receives a request:
  - If **no** other thread is in the critical section, it sends back a *reply* message
  - If another thread **is** in the critical section, the coordinator adds the request to the tail of its queue, and does not respond
- When the requesting thread receives the *reply* message from the coordinator, it enters the critical section
  - When it leaves the critical section, it sends a *release* message to coordinator
  - When the coordinator receives a *release* message, it removes the request from the head of the queue, and sends a *reply* message to that thread

Central Coordinator (cont.)



- Evaluation:
  - 3 messages required to enter CS
    - release, request, reply
  - ✘ Coordinator is a performance bottleneck
  - ✘ Coordinator is a single point of failure
  - ✘ Delay is unconstrained

Lamport's Algorithm (1978)

- Each process maintains a request queue, ordered by timestamp value
- Requesting the critical section (CS):
  - When a thread wants to enter the CS, it:
    - Adds the request to its own request queue
    - Sends a timestamped *request* message to all threads in that CS's request set
  - When a thread receives a *request* message, it:
    - Adds the request to its own request queue
    - Returns a timestamped *reply* message
- Executing the CS:
  - A thread enters the CS when **both**:
    - Its own request is at the top of its own request queue (its request is earliest)
    - It has received a *reply* message with a timestamp larger than its request from all other threads in the request set

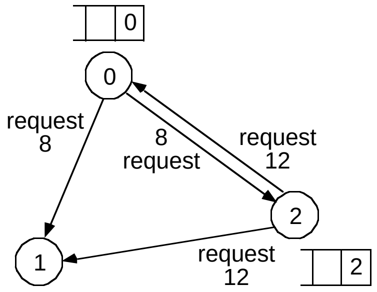
---

Lamport's Algorithm (cont.)

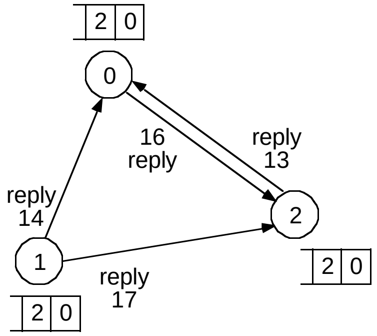
- Releasing the CS:
  - When a thread leaves the CS, it:
    - Removes its own (satisfied) request from the top of its own request queue
    - Sends a timestamped *release* message to all threads in the request set
  - When a thread receives a *release* message, it:
    - Removes the (satisfied) request from its own request queue
    - (Perhaps raising its own message to the top of the queue, enabling it to finally enter the CS)
- Evaluation:
  - 3(N–1) messages required to enter CS
    - (N–1) release, (N–1) request, (N–1) reply
  - ✘ Later…

Lamport's Algorithm (cont.)

- Both threads 0 and 2 request the CS:



- Everyone replies, thread 0 enters the CS since its request was first:



Lamport's Algorithm (cont.)

- Thread 0 releases the CS, thread 2 enters it: