Synchronization

527950-1
Fall 2019
11/14/2019
Kyoung Shin Park
Applied Computer Engineering
Dankook University

Mutual Exclusion

- **Mutex = mutual exclusion**
- "ensure that multiple process that share resources do not attempt to share the same resource at the same time"
- "The concurrent access to a shared resource by several uncoordinated user-requests is serialized to secure the integrity of the shared resource"
- How can this be accomplished for distributed system?

Critical Section

- A critical section (CS) is "a code segment in which a shared resource is referenced" and "the portion of code or program accessing a shared resource"
- We must prevent concurrent execution by more than one process at a time
  - **Mutual exclusion** is one part of the solution to this problem
- Requirements:
  - If no process is in the critical section, **any requesting process may access** it without delay
  - When two or more process want the critical section, a section of which process to enter the critical section **cannot be delayed indefinitely**
  - **No process can prevent** another process from entering the critical section **indefinitely**

Critical Section Problem

- Consider a system consisting of n processes {P_0, P_1, P_2, .. P_n}
  - Each process has a segment of code, called a **critical section**
  - The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section
  - The execution of critical sections is **mutually exclusive** in time
- Assume we have several sequential processors
  - Let these processors share a common data area (i.e., data within a shared memory)
  - This is the same as multi-programmed processes or cooperating sequential processes

Critical Section Problem

- Assume that each process has the following code:

```
P_i() {
    while (true) {
            criticalSection_i;
            program_i; // remainder section
    }
}
```

  - Within **criticalSection_i**, the common data area is referenced (i.e., data within the shared memory)
  - Only one process can be in its critical section at any given time
  - Such processes may be called **loosely connected processes**. They are almost independent.

Critical Section Problem

- Assumptions:
  1. Only **one process** can **access** the shared memory **at a time**, simultaneous references would result in sequential accesses in an unknown order
  2. There is no priority among the critical sections
  3. Processors can be of different speeds (We don't want to depend on timing tricks)
  4. A process can halt, only if it is **not** in its critical section
- Now assume that the process all begin at the same time

```
begin
    P_1 and P_2 and P_3 and ... and P_n
end
```

Critical Section Solutions

- Critical Section Solutions
  - The solution must **ensure the two processes do not enter critical regions at the same time**
  - The solution must **prevent interference from processes not attempting to enter their critical regions**
  - The solution must **prevent starvation**
- Critical Section Solutions, alternate statement
  - A solution to the critical section problem must show that
    - **Mutual exclusion is preserved**
    - **Progress requirement is satisfied**
    - **Bounded-waiting requirement is met**

Critical Section Solutions

- How is **mutual exclusion** preserved?
  - If **process P_1 is executing** in its critical section, then no other processes can be executing in their critical sections
- How is the **progress** requirement satisfied?
  - If no process is executing in its critical section and there exist **some processes that wish to enter** their critical sections, then only those processes that are not executing in their remainder section **can participate** in the decision of which will enter its critical section next, and this section **cannot be postponed indefinitely**
- How is **bounded waiting** met?
  - There exists a **bound on the number of times** that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical Section Hardware Solutions

- Hardware mechanism
  - Needed are some **atomic(i.e., non-interruptible)** hardware operations
    - That atomically test if a critical section is currently occupied
    - That can grab it if it not busy
    - And otherwise will wait, while continuing to test it until it is not busy
  - Possible solutions include
    - Atomic hardware operations
    - Hardware locks
    - Spin locks
    - Mutex hardware operations
    - Cache concurrency control

Critical Section Hardware Solutions

- In a uniprocessor,
  - The CS can disable/enable interrupts whenever a process enters/exits the CS
- In a multiprocessor or a distributed system,
  - The uniprocessor solution won't work since it can only affect the machine on which the CS lives
  - We need to use **test-and-set mechanisms**, **busy-waiting**, **atomic swaps** or **some software solutions**
- While **atomic operations** are difficult to implement in parallel computers (multiprocessors),
  - They are possible (e.g. spinlocks in the Encore Multimax)
- However, for distributed systems
  - The solutions need to done in software

Critical Section Hardware Solutions

- **Lock mechanisms**
  - A **lock** is one form of hardware support for mutual exclusion
  - If a shared resource has a locked hardware lock, it is already in use by another process
  - If it is not locked, a process may freely
    - **Lock it for itself**
    - **Use it**
    - **Unlock it when it finishes**
  - Problem: Race conditions

만약 여기서 interrupt 되면

```
lock(lock);
--critical section
unlock(lock);
--remainder section
```

```
struct lock {int held =0;}
void lock(struct lock *l) {
    while (l -> held);
    l->held =1;
}
void unlock (struct lock &l) {
   l-> held =0;
}
```

Critical Section Hardware Solutions

- **Test-and-Set**
  - Is a hardware implementation for testing for the lock and resetting it to locked
  - If test shows unlocked, the process may proceed
  - Acts as an **atomic operation**
  - Permits
    - Busy waits
    - Spinning
    - Spinlocks

```
do {
  while(test_and_set(&lock));
   --critical section
  lock = 0;
   --remainder section
} while(true);
```

```
int test_and_set(int *target) {
  int rv = *target;
  *target = 1;
  return rv;
}
```

Critical Section Hardware Solutions

- **Atomic Swap**
  - Performs three operations **atomically**
    1. Swap current lock value with temp locked lock
    2. Examine new value of temp lock
    3. If locked, repeat
       If unlocked, proceed into critical section
  - Utilizes a temporary variable

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;
  if (*value = expected) {
    *value = new_value;
  }
  return temp;
}
```

```
do{
  while(compare_and_swap(&lock, 0, 1) !=0);
   --critical section
  lock = 0;
   --remainder section
}while(true);
```

Critical Section Software Solutions

- Software mechanisms
  - Possible software solutions include
    - Software locks
    - Programming language constructs
      - **Semaphores**
      - **Critical regions**
      - **Monitors**
    - System library support
  - In distributed systems mechanisms for critical section software solutions also include
    - **Centralized lock manager algorithms**
    - **Distributed lock manager algorithms**
    - **Token-passing algorithms**
    - **Election algorithms**

Critical Section Software Solutions

- Every algorithm used for software mutual exclusion solutions must meet the following criteria
  - Does **mutual exclusion** hold?
  - Is **interference** from other processes not currently trying to get into the critical section **prevented**?
  - Are all processes waiting for the shared resource (or critical section) **protected against starvation**? That is, will each process **eventually be scheduled**?

Centralized Lock Manager

- A centralized lock manager maintains information on
  - Which processes have requested access to critical section
  - Which processes have been granted request
- A centralized lock manager algorithm:
  - **Request** message (required)
    - When a process needs to access CS, it sends a **request message** to the centralized lock manager, requesting entry to CS
  - **Granted** message (required)
    - If CS is available, the lock manager returns a **granted message** to the requesting process
  - **Queued** message (**optional**)
    - If CS is not available, the lock manager may return a **queued message (wait)**, in some versions of this type of algorithm
  - **Release** message (required)
    - When a process that has been granted access to a CS finishes CS, it sends a **release message** back to the lock manager so that another process can be granted access

Centralized Lock Manager

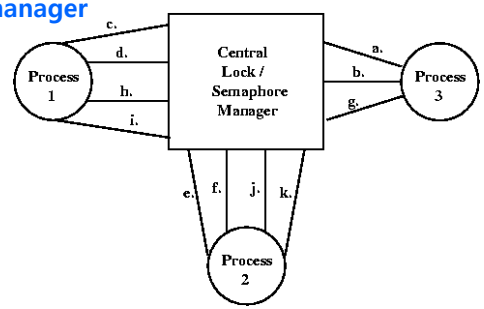
- Correctness of the algorithm
  - Does the algorithm assure **mutual exclusion**?
    - **Yes**, as only one process is granted access at a time, and not other process can enter its critical section until the first process finishes and sends a release message
  - Can **other processes** not in competition for the critical section **interfere** with access to the critical section?
    - **No**, as only processes requesting or using the critical section communicate with the lock manager
  - Is **starvation** possible?
    - **No**, as unfulfilled requests are queued until granted access

Centralized Lock Manager

- Problems/Disadvantages
  - The lock manager is a single critical component
    - There is no redundancy. The single lock manager may crash and bring down entire system.
    - If it goes down, all processes dependent on it may go down as well
  - This means there is **a single point of failure**
    - This would be unusable for real-time systems
  - Since all messages are sent to the lock manager or received from the manager
    - There is **increased traffic** to/from that node
    - This creates a possible bottleneck in the network

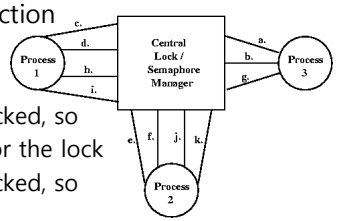
Centralized Lock Manager

- Example
  - Now consider a distributed system with three running process, Process1, Process2, Process3
  - Each of the processes has access to a shared data
  - Each process has a critical section
  - Assume the **mutual exclusion** is controlled by **a centralized lock manager**



Centralized Lock Manager

- Initially no processes are in their critical section



a. Process 3 **request** its CS and
b. Process 3 **granted** access to its CS
c. Process 1 **requests** its CS but the CS is locked, so
d. Process 1 is placed on a **queue waiting** for the lock
e. Process 2 **requests** its CS but the CS is locked, so
f. Process 2 is **queued**
g. Process 3 finishes its CS, **releasing** access to the CS and opening the lock
h. The central manager checks the lock, finding it unlocked and Process 1 is **granted** access to it CS by the central manager
i. Process 1 finishes it CS **releasing** access to the CS and opening the lock
j. The central manager checks the lock, finding it unlocked and Process 2 is **granted** access to its critical section by the central manager
k. Process 2 finishes its CS **releasing** access to the CS and opening the lock

Distributed Lock Manager

- A distributed lock manager avoids both the central point of failure and the network traffic hotspot problems
- A distributed lock manager algorithm:
  - **Request** message (required)
    - When a process needs to access its critical section, it sends a **request** message with its current **timestamp to all the other processes**, requesting entry to the critical section. This may be done as a broadcast message or as a set of individual messages.
  - **Queued** message (**optional**)
    - If another process is in the critical section, that process queues the request and may return a **queued** message, in some versions of this type of algorithm. This optional message helps distinguish between those processes that are very busy with other requests and those that have died.

Distributed Lock Manager

  - **Granted** message (required)
    - If any of the other processes are not in the critical section and have no other request with an earlier timestamp waiting in their queue, they return a **granted** message to the requesting process
    - If another process is in its critical section, it queues the requests of all other processes until it is done. Then it sends out a **granted** message **to every process in its queue**, much like a release message.
    - This means that a process may enter its critical section only if it has received **granted** messages (or votes) **from the majority of the other processes**. (Not absolute permission - Note that it is not necessary to receive an unanimous vote.)

Distributed Lock Manager

- Correctness of the algorithm
  - Does the algorithm assure **mutual exclusion**?
    - **Yes**, as only one process is granted access at a time by the majority of all other process.
    - Two or more processes cannot receive a majority of the votes at the same time
  - Can **other processes** not in competition for the critical section **interfere** with access to the critical section?
    - **No**, as it only requests a vote from a majority of the other processes.
    - However, without the queued message, it is possible that a large number of failed processes could prevent a process from proceeding into its critical section.
  - Is **starvation** possible?
    - **No**, as unfulfilled requests are queued in order of timestamp until granted access.

Distributed Lock Manager

- Problems/Disadvantages
  - While the lock manager is no longer a single critical component, all processes now take part in granting a critical section request. If **a majority of the processes fail**, the other processes could **deadlock** waiting for a decision.
  - This is one reason the **queued messages** are used. If a process never receives a queued message from another given process (perhaps after several request), it knows that process must be dead. Thus the number of processes needed for a majority vote may be reduced.
  - Furthermore, far more messages are generated by this algorithm. As a process leaves its critical section, it sends out granted messages to all process quests in its queue. These same processes may send out queued messages as responses to all request messages as well.
  - This means we have **far more traffic on the network** as a whole, instead of just one network hotspot.

Lamport's Mutual Exclusion Algorithm

- Each process maintains request queue
  - Queue contains **mutual exclusion requests**
  - Messages are sent reliably and in FIFO order
  - Each message is **timestamped** with totally ordered Lamport timestamps
    - Ensures that each timestamp is unique
    - Every node can make the same decision by comparing timestamps
  - Queues are sorted by message timestamps

Lamport's Mutual Exclusion Algorithm

- Request a critical section
  - Process Pi sends **request(i, Ti)** to all nodes
    - And places request on its own queue
  - When a process **Pj** receives a request
    - It returns a timestamped **ACK**
    - Places the request on its request queue

Process	Timestamp
P4	1021
P8	1022
P1	3944
P6	8201
P12	9638

- Enter a critical section (accessing resource)
  - Pi has received ACKs from everyone
  - Pi's request has the earliest timestamp in its queue
- Release a critical section
  - Process Pi removes its request from its queue
  - Sends **release(i, Ti)** to all nodes
  - Each process now checks if its quest is the earliest in its queue – It so, that process now has the critical section.

Lamport's Mutual Exclusion Algorithm

- N points of failure
- A lot of messaging traffic
  - Requests & releases are sent to the entire group

Ricart & Agrawala's Algorithm

- Distributed mutual exclusion algorithm using reliable multicast and logical clocks (Ricart & Agrawala 1981)
- When a process wants to enter critical section,
  1. Compose message containing
     - Identifier (machine ID, process ID)
     - Name of resources
     - Timestamp (e.g., totally-ordered Lamport)
  2. Reliably multicast request to all processes in group
  3. Wait until everyone gives permission
  4. Enter critical section & use resources

Ricart & Agrawala's Algorithm

- When a process receives request,
  - If receiver **not interested**
    - **Send OK** to sender
  - If receiver is **in critical section**
    - **Do not reply**, add request to queue
  - If receiver just sent a request as well (**potential race condition**)
    - **Compare timestamps** on received & sent messages
    - **Earliest wins**
    - If receiver is loser, send OK
    - If receiver is winner, do not reply, queue it
- When done with critical section
  - **Send OK** to all queued requests

Ricart & Agrawala's Algorithm

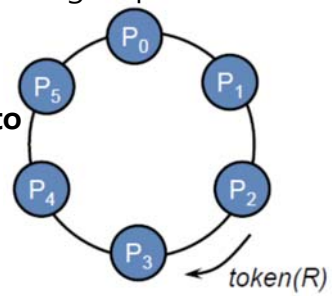
- Not great either
  - N points of failure
  - A lot of messaging traffic
  - Also, demonstrates that a fully distributed algorithm is possible

Lamport vs Ricart & Agrawala Algorithm

- Lamport
  - Everyone responds (ACKs) always – no hold-back
  - 3(N-1) message
    - Request – ACK – Release
  - Process decides to go based on whether its request is the earliest in its queue
- Ricart & Agrawala
  - If you are in the critical section (or won a tie)
    - Don't respond with an ACK until you are done with the critical section
  - 2(N-1) message
    - Request - ACK
  - Process decides to go if it gets ACKs from everyone

Token-Passing Mutual Exclusion

- There are many **token-passing algorithms** that may be used on parallel or distributed systems
- The network of processors must be logically **arranged as a ring** of processors.
- There is only one unidirectional path through the ring.
- **Only one token is active** in a logical ring of processes.
- When **a process holds the token**, it may **enter its critical section**. **Otherwise, it passes the token onto the next** logical process in the ring.
- **When the process is done** with its critical section, it **passes the token onto the next** process.



Token-Passing Mutual Exclusion

- Correctness of the algorithm
  - Assure **mutual exclusion**?
    - **Yes**, there is only **one token**.
    - Only one process can hold the token at a time.
  - **No interference** from processes not needing mutex?
    - **Yes**, if a process does not need mutual exclusion, it simply passes the token on.
  - Possible **starvation**?
    - **No**, a process holding the token may enter mutual exclusion only once.
    - Then, it must pass the token on to the next process.

Token-Passing Mutual Exclusion

- Complications
  - What if **the token gets lost**?
    - How might that happen?
    - How can you tell if that happens?
    - How do you know that the token is not just being used for a long time?
  - What if the process **holding token crashes**?
  - What if **some other process crashes**?
    - Is the ring destroyed?
  - How can we **add (and identify)** other processes to the ring?
  - How can we **remove some processes** from the ring?

Token-Passing Mutual Exclusion

- Possible Solutions
  - Make one centralized node be a **monitor (synchronized object)** and/or make the token a **monitor**
    - Send messages to **request the current location of the token**
    - If the token is missing, **start a new token**
    - **All new nodes** (and recovered nodes) **should check** in with the monitor
    - **Lost nodes can be recorded in the monitor** via the neighbors of the lost nodes
  - Make all nodes be **monitors**

Tree-Based Token Algorithm for Mutual Exclusion

- This algorithm is a distributed algorithm, implemented with a global FIFO queue.
  - **Local FIFO queues** are linked to form a **global queue** using a **tree topology**.
  - Recall in the Token-Ring algorithm that the token was passing along around the ring even when no node wanted or needed it.
- An alternate approach would be as follows
  - A **process requests the token** whenever it needs it
  - The token only moves to another node when it is requested.
  - The process's request message must find the node on which the token is located.
- In the topology of a tree, other nodes can always find the root by traveling back up through their ancestors.

Tree-Based Token Algorithm for Mutual Exclusion

- The rules are quite simple.
  - The **token** is always located at the **root** of the tree.
  - The tree is considered a **directed graph** with the arcs all pointing toward the root.
  - Each node of the tree represents the current logical location of a process.

Tree-Based Token Algorithm for Mutual Exclusion

- The algorithm works as follows
  1. Whenever a node's process wants **mutual exclusion (the token)** it sends the **request** to the **root** of the tree.
  2. When the token is **acquired**, it must first move to the requesting node. In doing so, it reshapes the tree so that the **requesting node becomes the root** of the tree.
  3. Each process maintains, a **FIFO queue of requests** and a pointer to its parent in the tree.
  4. Whenever a **request** is made, it is **appended to the FIFO queue** at that node.
  5. If the node does **not** currently have the **token** and the **queue** is **empty**, then node (process) must send the **request** to its **parent** node.
  6. If the node does have the **token** and the FIFO **queue** is **not empty**, the process **removes the top entry** in the queue and **sends the token to that node** (process), while changing the direction of its own pointer to point toward the process when the token is sent.

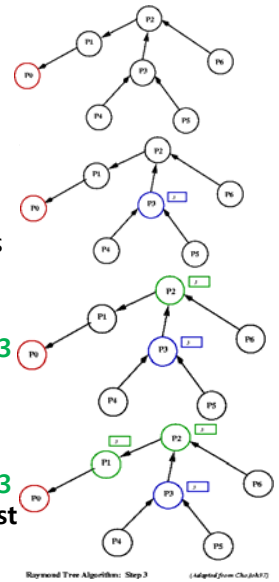
Tree-Based Token Algorithm for Mutual Exclusion

- Correctness of the algorithm
  - Assure **mutual exclusion**?
    - **Yes**, there is only **one token**.
    - Only one process can hold the token at a time.
  - **No interference** from processes not needing mutex?
    - **Yes**, a token only moves to those nodes that request it (mutual exclusion).
  - Possible **starvation**?
    - **No**, a process wanting the token will be placed in a FIFO queue, and it will eventually be granted the token.

Recall that a tree is just a acyclic graph, meaning that not possible to form a cyclic wait. This prohibits deadlock.

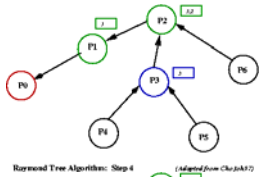
Tree-Based Token Algorithm for Mutual Exclusion

- Example
  1. Initially, **P0** holds the token. Also, P0 is the current **root**.
  2. **P3** wants the token to get into its critical section. So, P3 adds **itself** to its **own FIFO queue** and sends a **request** message to its **parent P2**.
  3. **P2** receives the request from P3. It **adds P3 to its FIFO queue** and passes the **request** message to its parent **P1**.
  4. **P1** receives the request from P2. It **adds P3 into its FIFO queue** and passes the **request** message to its parent **P0**.

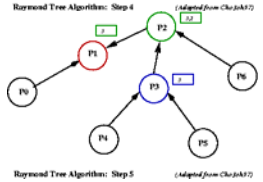


Tree-Based Token Algorithm for Mutual Exclusion

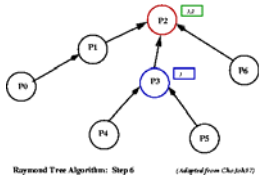
5. At this point, **P2** also **wants the token**. Since its FIFO queue is not empty, it adds **itself** to **its own FIFO queue**.



6. P0 receives the request from **P3** through **P1**. **P0 surrenders** the token and passes it on to **P1**. It also changes the direction of the arrow between them, making **P1 the root**, temporarily.

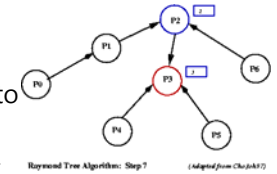


7. P1 removes the top element of its FIFO queue to see which node requested the token. Since the token needs to go to P3, **P1 surrenders** the token and passes it onto **P2**. It also changes the direction of the arrow between them, making **P2 the root**, temporarily.

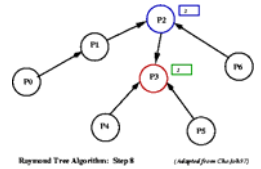


Tree-Based Token Algorithm for Mutual Exclusion

8. **P2** removes the top element of its FIFO queue to see which node requested the token. Since the token needs to go to P3, **P2 surrenders** the token and passes it onto **P3**. It also changes the direction of the arrow between them, making **P3 the root**.

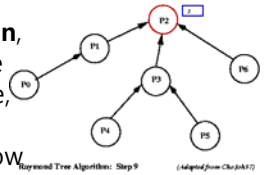


9. Now, **P3 holds the token and can execute its critical section**. It is able to clear the top (and only) elements of its FIFO queue. Note that P3 is the current root.
10. In the meantime, **P2** checks the top element of its FIFO queue and realizes that is also needs to request the token. So, P2 sends a **request** message to its current **parent**, **P3**, who **appends the request to its FIFO queue**.

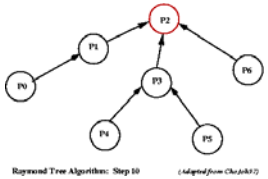


Tree-Based Token Algorithm for Mutual Exclusion

11. As soon as **P3 completes its critical section**, it checks the top element of its FIFO queue to see if it is needed elsewhere. In this case, P2 has requested it, so P3 sends it back to P2. It also changes the direction of the arrow between them, making **P2 the new root**.



12. **P2** holds the token and is able to complete its critical section. Then it checks it FIFO queue, which is empty. So it **waits until** some other node **request** the token.



Election Algorithms

- Many of the algorithms for distributed systems require some centralization, some site with **leader/coordinator** activities.
- All other sites need to recognize this leader; often there is accomplished by an initial **agreement/election**.
- When the site of the leader fails or goes down for some reason, it is necessary to **elect a new leader**.
- This is the purpose of election or agreement algorithms.

Election Algorithms

- There are two basic criteria for an election algorithm.
  - One way to decide the leader is to use some global priority.
    - The **Bully algorithm** by Garcia-Molina (1982)
  - The second is a more general, preference-based algorithm, that permits some nodes to have heavier votes.
    - **Token-Ring election algorithm** by Chang & Robert (1979)

Election Algorithms

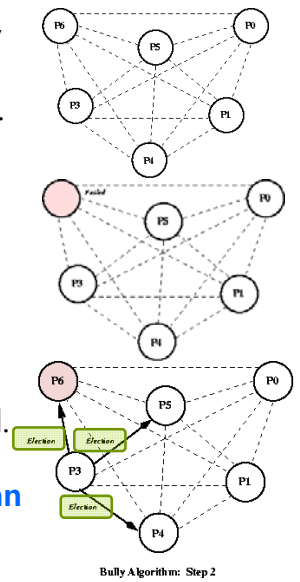
- Assumptions for most election algorithms
  - A complete topology, i.e., one message hop between any two processes
  - All **process IDs** are **unique** and known to all other processes.
  - All **communication networks** are **reliable**, i.e., only communicating processes may fail.
  - This assures that no messages are
    - Lost
    - Duplicated
    - Corrupted
  - A recovering process is aware that it failed
    - Failure is reliably detected by setting the time-out interval to be a little larger than the sum of the round-trip message delay and the message processing time.
    - A failed process can rely on the coordinator to poll periodically for recovered process so that they may rejoin the pool of processes.

Bully Election Algorithm

- The Bully Election Algorithm (Garcia-Molina)
  - One process notices that the leader/server is missing and
    - Sends messages to all other processes
    - Requests to be appointed leader
    - Includes his processor number
  - Processes with **higher** (lower) **process numbers** can **bully** the first process.
  - The process with **highest ID wins the election** and sends out a message to that effect.
  - The process that initiates the election need only send messages about **election to higher numbered processes**.
  - Any processes that **respond** effectively tell the first process that they overrule him and that he is out of the running as they have higher IDs.
  - These processes then start sending **election** messages to the other high-number processes.

Bully Election Algorithm

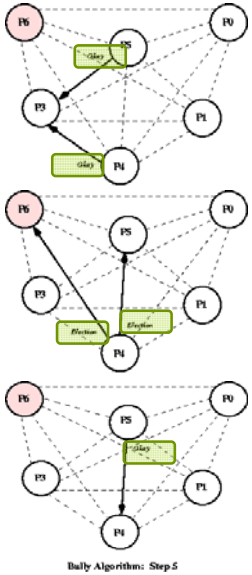
1. We start with 6 processes, all directly connected to each other. **P6 is the leader**, as it has the **highest** number.
2. P6 fails
3. **P3 notices** that P6 does not respond. So it starts an **election**, **notifying those processes with IDs grater than 3**.



Bully Algorithm: Step 2

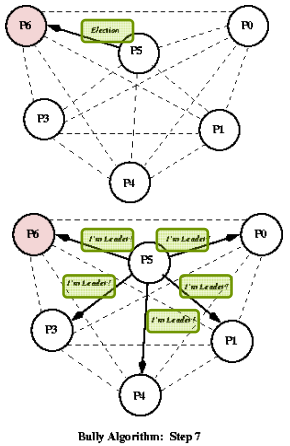
Bully Election Algorithm

4. Both **P4 and P5 respond**, telling P3 that they'll take over from here.
5. **P4** sends **election** messages to both **P5 and P6**.
6. **Only P5 answers** and takes over the election.



Bully Election Algorithm

7. **P5** sends out only one **election** message to **P6**.
8. When **P6** does **not respond**, **P5 declares itself the winner.**

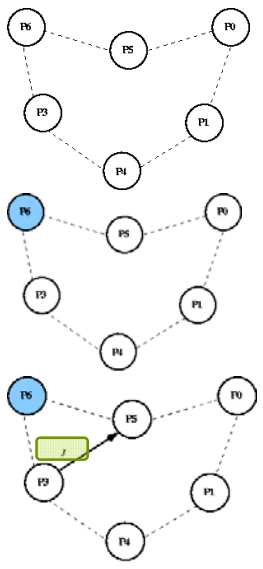


Token-Ring Election Algorithm

- Token-Ring Election Algorithm (Chang & Roberts)
  - Each process has a **unique ID**. **Each process knows its successor in the ring**.
  - When **a process notices the leader is down**, it sends an **election** message to **its successor**.
  - If **the successor is down**, the originating process sends the message to **the next process** in the logical ring.
  - Each process that receives an election message, passes it on to the next process in the ring. **Each sender appends its own ID** to the message.
  - The election message eventually **returns to the originating process** and contains its ID.
  - At this point, the election message is changed to a **coordinator (new leader) message and sent around ring**. The process with **the highest ID** in the circulated election message becomes the new leader.
  - When **the coordinator message comes back to the originating process**, it is **deleted**.

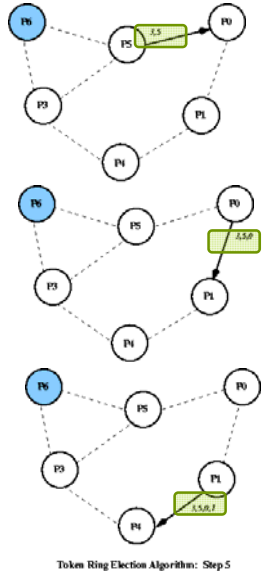
Token-Ring Election Algorithm

1. We start with 6 processes, connected in a logical ring. **P6** is the **leader**, as it has the **highest number**.
2. P6 fails
3. **P3 notices** that P6 does not respond. So it starts an **election**, sending a message containing **its ID** to **the next node** in the ring.



Token-Ring Election Algorithm

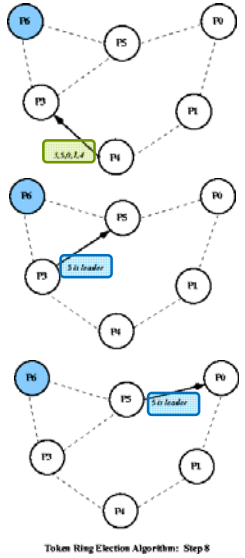
4. **P5** passes the message on, **adding its own ID** to the message.
5. **P0** passes the message on, **adding its own ID** to the message.
6. **P1** passes the message on, **adding its own ID** to the message.



Token Ring Election Algorithm: Step 5

Token-Ring Election Algorithm

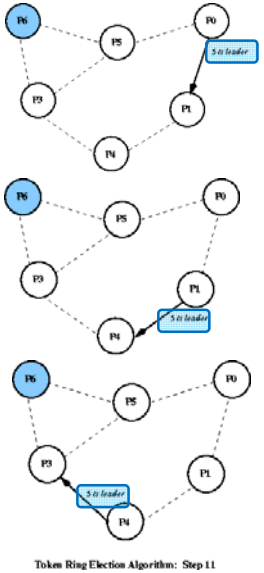
7. **P4** passes the message on, **adding its own ID** to the message.
8. When **P3 receives the message back**, it knows the message has done around the ring, as it own ID is in the list. **Picking the highest ID** in the list, it starts **the coordinator message "5 is the leader"** around the ring.
9. **P5** passes on the coordinator message.



Token Ring Election Algorithm: Step 8

Token-Ring Election Algorithm

10. **P0** passes on the coordinator message.
11. **P1** passes on the coordinator message.
12. **P4** passes on the coordinator message.
13. **P3** receives the coordinator message, and **stop it**.



Token Ring Election Algorithm: Step 11

References

- http://www.cs.colostate.edu/~cs551/CourseNotes/Synchronization/SynchTOC.html
- https://www.cs.rutgers.edu/~pxk/417/notes/10-mutex.html