**LAB : 6**

**Bully Algorithm:**

The Bully Algorithm is a leader election algorithm used in distributed computing systems where processes need to elect a coordinator or leader among themselves. It ensures that if the current coordinator fails, a new one is elected to take its place. Here's how it works:

1. **Process Hierarchy:** Each process in the system has a unique ID or rank, and processes are organized in a hierarchy based on their IDs.
2. **Election Trigger:** If a process detects that the coordinator has failed (e.g., due to a timeout), it initiates an election by sending an election message to all processes with higher IDs.
3. **Election Process:**
   * When a process receives an election message, it responds by sending an "OK" message to the sender.
   * If a process does not receive any response within a timeout period, it declares itself as the coordinator and sends a message to all processes with lower IDs, informing them about its new role.
   * If a process receives a response from a higher-ranked process, it aborts its election process.
4. **Coordinator Confirmation:** Once a new coordinator is elected, it broadcasts a coordinator message to inform all processes about its role.
5. **Handling Coordinator Failure:** If a process receives a coordinator message from a higher-ranked process, it updates its local coordinator variable. This ensures that if the current coordinator fails, a new one is quickly elected.

**Ring Algorithm:**

The Ring Algorithm is another leader election algorithm used in distributed computing systems. It operates on a logical ring topology, where processes are connected in a ring structure. Here's how it works:

1. **Ring Topology:** Processes are organized in a logical ring structure, where each process has knowledge of its neighboring processes.
2. **Election Trigger:** When a process detects that the coordinator has failed or when a new process joins the system, it initiates an election by sending an election message to its neighbor(s).
3. **Election Process:**
   * Upon receiving an election message, a process compares its own ID with the ID in the message.
   * If the ID in the message is higher, the process forwards the message to its neighbor(s).
   * If the ID in the message is lower or equal, the process replaces the ID in the message with its own ID and forwards the modified message to its neighbor(s).
4. **Coordinator Confirmation:** Once a process receives its own ID in the election message, it knows that it has completed the ring and has the highest ID. It declares itself as the coordinator and sends a coordinator message in the opposite direction to inform all other processes about its new role.
5. **Handling Coordinator Failure:** If a process detects that the coordinator has failed, it triggers a new election process by initiating an election message to its neighbor(s), restarting the process described above.

**Comparison:**

* **Topology:** Bully Algorithm doesn't assume any specific topology, while the Ring Algorithm operates on a logical ring topology.
* **Message Complexity:** Bully Algorithm may involve more message exchanges compared to the Ring Algorithm because it sends messages to all higher-ranked processes, whereas the Ring Algorithm only sends messages to neighboring processes.
* **Fault Tolerance:** Bully Algorithm is better suited for systems where failures are common, as it quickly elects a new coordinator when the current one fails. The Ring Algorithm may take longer to detect and elect a new coordinator, especially if the failure occurs far from the initiator.
* **Implementation Complexity:** Ring Algorithm's implementation can be simpler as it relies on the logical ring structure and simple message forwarding logic, whereas Bully Algorithm may involve more complex logic for handling multiple election messages and coordinator confirmations.

import java.util.Scanner; public class Bully { int coordinator; int max\_processes; boolean processes[];

* This part imports the Scanner class for user input and defines a class named **Bully**.
* It declares instance variables **coordinator**, **max\_processes**, and **processes[]**, which store the current coordinator process ID, the maximum number of processes, and an array to track the status of each process.

java

Copy code

public Bully(int max) { max\_processes = max; processes = new boolean[max\_processes]; coordinator = max;

* This is the constructor method **Bully**, which initializes the **max\_processes** and **processes[]** array.
* It sets the initial coordinator as the process with the highest ID (assuming IDs start from 1 and are consecutive).

java

Copy code

System.out.println("Creating processes.."); for(int i = 0; i < max; i++) { processes[i] = true; System.out.println("P"+ (i+1) + " created"); } System.out.println("Process P" + coordinator + " is the coordinator"); }

* Inside the constructor, it initializes each process in the **processes[]** array as active (true) and prints a message for each process creation.
* It also prints the initial coordinator process ID.

java

Copy code

void displayProcesses() { for(int i = 0; i < max\_processes; i++) { if(processes[i]) { System.out.println("P" + (i+1) + " is up"); } else { System.out.println("P" + (i+1) + " is down"); } } System.out.println("Process P" + coordinator + " is the coordinator"); }

* This method **displayProcesses()** prints the status of each process (up/down) and the current coordinator.

java

Copy code

void upProcess(int process\_id) { if(!processes[process\_id - 1]) { processes[process\_id - 1] = true; System.out.println("Process " + process\_id + " is now up."); } else { System.out.println("Process " + process\_id + " is already up."); } } void downProcess(int process\_id) { if(!processes[process\_id - 1]) { System.out.println("Process " + process\_id + " is already down."); } else { processes[process\_id - 1] = false; System.out.println("Process " + process\_id + " is down."); } }

* These methods **upProcess(int process\_id)** and **downProcess(int process\_id)** handle bringing processes up and down respectively.
* They update the status of the specified process in the **processes[]** array and print appropriate messages.

java

Copy code

void runElection(int process\_id) { coordinator = process\_id; boolean keepGoing = true; for(int i = process\_id; i < max\_processes && keepGoing; i++) { System.out.println("Election message sent from process " + process\_id + " to process " + (i+1)); if(processes[i]) { keepGoing = false; runElection(i + 1); } } }

* This method **runElection(int process\_id)** initiates the election process starting from the specified process ID.
* It sends election messages to higher-numbered processes and recursively calls itself with the next process ID until a live process is found.

java

Copy code

public static void main(String args[]) { Bully bully = null; int max\_processes = 0, process\_id = 0; int choice = 0; Scanner sc = new Scanner(System.in); while(true) { // Display menu options

* The **main** method starts the program execution.
* It declares variables for user input and initializes a Scanner object.
* It enters a loop to continuously display menu options and handle user input.

java

Copy code

switch(choice) { case 1: // Create processes // Prompt user for the number of processes // Create a new Bully object with the specified number of processes break; case 2: // Display processes // Call the displayProcesses() method of the Bully object break; case 3: // Up a process // Prompt user for the process number to bring up // Call the upProcess(int process\_id) method of the Bully object break; case 4: // Down a process // Prompt user for the process number to bring down // Call the downProcess(int process\_id) method of the Bully object break; case 5: // Run election algorithm // Prompt user for the process number to initiate election // Call the runElection(int process\_id) method of the Bully object // Display processes after election break; case 6: // Exit program // Terminate the program break; default: // Handle invalid choice // Display error message break; }

* This part of the **main** method handles user choices by utilizing a switch-case structure.
* It executes different actions based on the user's choice, such as creating processes, displaying processes, bringing processes up/down, running the election algorithm, or exiting the program.
* Each case corresponds to a specific action and calls the appropriate method of the **Bully** object.

java

Copy code

} } }

* The **main** method ends here, and so does the class definition.

This part of the code is within the constructor of the **Bully** class. Let's break it down step by step:

java

Copy code

System.out.println("Creating processes..");

* This line simply prints the message "Creating processes.." to the console, indicating that the process of creating processes is beginning.

java

Copy code

for(int i = 0; i < max; i++) { processes[i] = true; System.out.println("P"+ (i+1) + " created"); }

* This **for** loop iterates from 0 to **max - 1**, where **max** is the maximum number of processes specified when creating a new instance of the **Bully** class.
* For each iteration, it sets the **i**th element of the **processes[]** array to **true**, indicating that the process with ID **i+1** is active.
* It also prints a message indicating the creation of each process, where **(i+1)** represents the process ID.

For example, let's say **max** is 5. The loop will create 5 processes with IDs 1, 2, 3, 4, and 5, and set their statuses as active. The output might look like this:

Copy code

Creating processes.. P1 created P2 created P3 created P4 created P5 created

java

Copy code

System.out.println("Process P" + coordinator + " is the coordinator");

* Finally, this line prints a message indicating which process is currently the coordinator.
* In the constructor, the coordinator is initially set as the process with the highest ID (**max**), assuming process IDs start from 1 and are consecutive.
* So, this line will print the ID of the coordinator process.

Continuing with the previous example where **max** is 5, the output might look like this:

arduino

Copy code

Process P5 is the coordinator

This section of the code initializes processes by setting them all as active and prints messages to indicate the creation of each process and which process is the initial coordinator.

This method **displayProcesses()** is responsible for showing the current status of each process, whether they are up or down, along with indicating the coordinator process. Let's dissect it:

java

Copy code

void displayProcesses() {

* This line declares a method named **displayProcesses()**. It doesn't return anything (**void**), indicating it's meant to perform some actions rather than compute a value.

java

Copy code

for(int i = 0; i < max\_processes; i++) {

* This line starts a **for** loop that iterates over each process in the **processes[]** array.

java

Copy code

if(processes[i]) { System.out.println("P" + (i+1) + " is up"); } else { System.out.println("P" + (i+1) + " is down"); }

* Within the loop, it checks the status of each process. If **processes[i]** is **true**, it means the process is up. It prints a message indicating that process **i+1** is up. If **processes[i]** is **false**, it means the process is down, and it prints a message indicating that process **i+1** is down.

For example, if we have **max\_processes** set to 3 and only the first process is up while the others are down, the output might look like this:

csharp

Copy code

P1 is up P2 is down P3 is down

java

Copy code

System.out.println("Process P" + coordinator + " is the coordinator");

* After looping through all processes, this line prints a message indicating which process is currently the coordinator.
* It simply prints the ID of the coordinator process.

Continuing with the previous example, if the coordinator process ID is 1, the output might look like this:

csharp

Copy code

P1 is up P2 is down P3 is down Process P1 is the coordinator

This method essentially provides a snapshot of the current state of all processes, indicating whether each is up or down, and which one is the coordinator.

These methods, **upProcess(int process\_id)** and **downProcess(int process\_id)**, are responsible for bringing a specific process up or down respectively. Let's break them down:

java

Copy code

void upProcess(int process\_id) {

* This line declares a method named **upProcess** which takes an integer argument **process\_id**. It doesn't return anything (**void**), indicating it's meant to perform actions rather than compute a value.

java

Copy code

if(!processes[process\_id - 1]) {

* This line checks if the process with the given **process\_id** is already up. It does so by accessing the **processes[]** array at index **process\_id - 1**.
* The **- 1** is used because array indices start from 0, but process IDs start from 1.

java

Copy code

processes[process\_id - 1] = true; System.out.println("Process " + process\_id + " is now up.");

* If the process is not already up, this block of code executes.
* It sets the status of the process with the given ID to **true**, indicating that the process is now up.
* It also prints a message indicating that the process with the given ID is now up.

java

Copy code

} else { System.out.println("Process " + process\_id + " is already up."); }

* If the process is already up, this block of code executes.
* It simply prints a message indicating that the process with the given ID is already up.

Similarly, let's discuss the **downProcess(int process\_id)** method:

java

Copy code

void downProcess(int process\_id) {

* This line declares a method named **downProcess** which takes an integer argument **process\_id**. Like the **upProcess** method, it doesn't return anything (**void**).

java

Copy code

if(!processes[process\_id - 1]) {

* This line checks if the process with the given **process\_id** is already down by accessing the **processes[]** array at index **process\_id - 1**.

java

Copy code

System.out.println("Process " + process\_id + " is already down.");

* If the process is already down, this block of code executes. It prints a message indicating that the process with the given ID is already down.

java

Copy code

} else { processes[process\_id - 1] = false; System.out.println("Process " + process\_id + " is down."); }

* If the process is not already down, this block of code executes.
* It sets the status of the process with the given ID to **false**, indicating that the process is now down.
* It also prints a message indicating that the process with the given ID is now down.

For example, if we call **upProcess(2)** to bring process 2 up, and then call **downProcess(3)** to bring process 3 down, the output might look like this:

arduino

Copy code

Process 2 is now up. Process 3 is down.

This indicates that process 2 has been brought up, and process 3 has been brought down.

This method **runElection(int process\_id)** initiates the Bully algorithm for leader election, starting from the given process ID. Let's break down its functionality:

java

Copy code

void runElection(int process\_id) {

* This line declares a method named **runElection** which takes an integer argument **process\_id**. It doesn't return anything (**void**), indicating it's meant to perform actions rather than compute a value.

java

Copy code

coordinator = process\_id;

* This line updates the coordinator variable to the given **process\_id**. In the Bully algorithm, when a process starts an election, it declares itself as the coordinator candidate. So, this line updates the coordinator candidate to the given process ID.

java

Copy code

boolean keepGoing = true;

* This line initializes a boolean variable **keepGoing** to **true**. It's used to control the loop.

java

Copy code

for(int i = process\_id; i < max\_processes && keepGoing; i++) {

* This line starts a **for** loop that iterates over processes with IDs greater than or equal to the given **process\_id**, up to the maximum number of processes (**max\_processes**), as long as **keepGoing** is **true**.

java

Copy code

System.out.println("Election message sent from process " + process\_id + " to process " + (i+1));

* This line prints a message indicating that an election message is being sent from the current process (the one initiating the election) to the process with ID **i+1**. The **i+1** is used because process IDs start from 1, while array indices start from 0.

java

Copy code

if(processes[i]) { keepGoing = false; runElection(i + 1); }

* Inside the loop, it checks if the process with ID **i+1** (the next process in line) is active.
* If the process is active, it means the current process has found a higher-ranked process. So, it sets **keepGoing** to **false** to stop the loop, and recursively calls **runElection** with the ID of the higher-ranked process.
* This recursive call initiates a new election starting from the higher-ranked process.

For example, if we call **runElection(3)** and process 3 sends election messages to processes 4 and 5, and process 5 responds indicating it's active, the output might look like this:

arduino

Copy code

Election message sent from process 3 to process 4 Election message sent from process 3 to process 5 Election message sent from process 5 to process 6 Process 6 is up

In this scenario, process 3 initiates the election, sends messages to processes 4 and 5. Process 5 responds, indicating it's up. Then, process 5 initiates a new election and finds that process 6 is up.