1.Program to implement non token based algorithm for Mutual Exclusion

```
#include <iostream>
#include <queue>
#include <vector>
#include <climits>
#include <thread>
#include <mutex>
#include <chrono>
#include <condition_variable>
using namespace std;
// Mutex and condition variable to simulate request queue
std::mutex mtx;
std::condition_variable cv;
int number_of_processes;
bool resource_available = true;
struct Request {
   int timestamp;
   int process_id;
};
// Global variables
std::vector<int> timestamps;
std::vector<bool> waiting;
std::queue<Request> request_queue;
// Function to compare two requests
bool compareRequest(Request a, Request b) {
   if (a.timestamp == b.timestamp)
      return a.process_id < b.process_id;
   return a.timestamp < b.timestamp;
}
// Function for a process to request access to a shared resource
void requestResource(int process_id) {
```

```
    std::unique_lock<std::mutex> lock(mtx);
    int timestamp = ++timestamps[process_id];
    cout << "Process " << process_id << " is requesting the resource with timestamp " << timestamp << endl;
    request_queue.push({timestamp, process_id});
    waiting[process_id] = true;
    // Wait until it's this process's turn to access the resource
    while (!resource_available || request_queue.front().process_id != process_id) {
        cv.wait(lock);
    }
    // Now process gets access to the resource
    resource_available = false;
    waiting[process_id] = false;
    cout << "Process " << process_id << " is accessing the resource." << endl;
    // Simulate the process holding the resource for some time
    std::this_thread::sleep_for(std::chrono::seconds(1));
    // Release the resource
    cout << "Process " << process_id << " has released the resource." << endl;
    resource_available = true;
    request_queue.pop();
    // Notify other waiting processes
    cv.notify_all();
}
// Simulate each process
void process(int process_id) {
    // Simulate some processing work
    std::this_thread::sleep_for(std::chrono::seconds(2 + process_id));
    // Request the shared resource
    requestResource(process_id);
}
int main() {
```

```
    number_of_processes = 3;  // You can modify this value
    timestamps.resize(number_of_processes, 0);
    waiting.resize(number_of_processes, false);
    std::vector<std::thread> threads;
    // Create threads for each process
    for (int i = 0; i < number_of_processes; ++i) {
        threads.push_back(std::thread(process, i));
    }
    // Join all threads
    for (auto& th : threads) {
        th.join();
    }
    return 0;
}
```

Sample Output

Process 0 is requesting the resource with timestamp 1

Process 1 is requesting the resource with timestamp 1

Process 2 is requesting the resource with timestamp 1

Process 0 is accessing the resource.

Process 0 has released the resource.

Process 1 is accessing the resource.

Process 1 has released the resource.

Process 2 is accessing the resource.

Process 2 has released the resource.

2.Program to implement Lamport's Logical Clock.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Function to simulate the maximum of two numbers
int max(int a, int b) {
    return (a > b) ? a : b;
}
// Class representing a process in the system
class Process {
public:
    int process_id;
    int clock;
    // Constructor
    Process(int id) {
        process_id = id;
        clock = 0; // Initialize the clock to 0
    }
    // Method to simulate an event happening in this process
    void event() {
        clock++;
        cout << "Event in Process " << process_id << " | Logical Clock: " << clock << endl;
    }
    // Method to send a message from this process to another process
    void send_message(Process& receiver) {
        clock++; // Increment the clock for the send event
        cout << "Process " << process_id << " sent a message to Process " << receiver.process_id << " |
Clock: " << clock << endl;
        receiver.receive_message(clock);
    }
```

```
    // Method to receive a message and update the clock
    void receive_message(int sender_clock) {
        clock = max(clock, sender_clock) + 1;
        cout << "Process " << process_id << " received a message | Updated Logical Clock: " << clock << endl;
    }
};
int main() {
    // Create processes
    Process p1(1);
    Process p2(2);
    Process p3(3);
    // Simulate events and message passing
    p1.event();       // Event in Process 1
    p2.event();       // Event in Process 2
    p1.send_message(p2); // Process 1 sends a message to Process 2
    p3.event();       // Event in Process 3
    p2.send_message(p3); // Process 2 sends a message to Process 3
    p1.event();       // Another event in Process 1
    return 0;
}
```

Sample Output

Event in Process 1 | Logical Clock: 1

Event in Process 2 | Logical Clock: 1

Process 1 sent a message to Process 2 | Clock: 2

Process 2 received a message | Updated Logical Clock: 3

Event in Process 3 | Logical Clock: 1

Process 2 sent a message to Process 3 | Clock: 4

Process 3 received a message | Updated Logical Clock: 5

Event in Process 1 | Logical Clock: 3

3.Program to implement edge chasing distributed deadlock detection algorithm.

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
class Process {
public:
   int id;  // Process ID
   vector<int> waitingFor;  // List of processes this process is waiting for
   Process(int id) : id(id) {}
};
// Function to detect deadlock using the Edge Chasing algorithm
bool detectDeadlock(vector<Process>& processes, int initiator) {
   queue<int> probeQueue;
   vector<bool> visited(processes.size(), false);
   // Initiator process starts the deadlock detection by sending probes
   probeQueue.push(initiator);
   visited[initiator] = true;
   cout << "Process " << initiator << " initiates deadlock detection." << endl;
   while (!probeQueue.empty()) {
      int currentProcess = probeQueue.front();
      probeQueue.pop();
      // Check if the current process is waiting for any other process
      for (int waitingFor : processes[currentProcess].waitingFor) {
         // If a probe returns to the initiator, a deadlock is detected
         if (waitingFor == initiator) {
            cout << "Deadlock detected: Process " << initiator << " is involved in a cycle!" << endl;
            return true;
         }
         // If the process has not been visited yet, send a probe to it
         if (!visited[waitingFor]) {
```

```
            visited[waitingFor] = true;
            cout << "Probe sent from Process " << currentProcess << " to Process " << waitingFor << endl;
            probeQueue.push(waitingFor);
        }
      }
    }
    // No deadlock detected if we exit the loop
    cout << "No deadlock detected." << endl;
    return false;
}
int main() {
    int numProcesses = 4;
    // Create processes with IDs 0, 1, 2, 3
    vector<Process> processes;
    for (int i = 0; i < numProcesses; i++) {
        processes.push_back(Process(i));
    }
    // Add dependencies (wait-for graph)
    // Example: Process 0 is waiting for Process 1, and Process 1 is waiting for Process 2, etc.
    processes[0].waitingFor = {1};  // Process 0 waits for Process 1
    processes[1].waitingFor = {2};  // Process 1 waits for Process 2
    processes[2].waitingFor = {3};  // Process 2 waits for Process 3
    processes[3].waitingFor = {0};  // Process 3 waits for Process 0, forming a cycle (deadlock)
    // Initiate deadlock detection from Process 0
    int initiator = 0;
    detectDeadlock(processes, initiator);
    return 0;
}
```

Sample Output

Process 0 initiates deadlock detection.

Probe sent from Process 0 to Process 1

Probe sent from Process 1 to Process 2

Probe sent from Process 2 to Process 3

Probe sent from Process 3 to Process 0

Deadlock detected: Process 0 is involved in a cycle!

4.Program to implement locking algorithm.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>  // For sleep_for
using namespace std;
// Global mutex to protect shared resources
mutex mtx;
// Shared resource (a counter)
int shared_counter = 0;
// Function to simulate critical section with locking using lock_guard
void increment_counter(int thread_id) {
    cout << "Thread " << thread_id << " is trying to acquire the lock..." << endl;
    // Lock the critical section using lock_guard (RAII: automatically locks and unlocks)
    lock_guard<mutex> lock(mtx);
    // Critical section starts
    cout << "Thread " << thread_id << " has acquired the lock." << endl;
    shared_counter++;
    cout << "Thread " << thread_id << " incremented the counter to " << shared_counter << endl;
    // Simulate some processing time
    this_thread::sleep_for(chrono::milliseconds(500));
    // lock_guard automatically releases the lock when going out of scope
    cout << "Thread " << thread_id << " has released the lock." << endl;
}
int main() {
    // Create multiple threads to access the shared resource
    thread t1(increment_counter, 1);
    thread t2(increment_counter, 2);
    thread t3(increment_counter, 3);
    // Wait for all threads to complete
    t1.join();
```

```
    t2.join();
    t3.join();
    // Final value of the shared resource
    cout << "Final value of shared_counter: " << shared_counter << endl;
    return 0;
}
```

Sample Output

Thread 1 is trying to acquire the lock...

Thread 1 has acquired the lock.

Thread 1 incremented the counter to 1

Thread 2 is trying to acquire the lock...

Thread 1 has released the lock.

Thread 2 has acquired the lock.

Thread 2 incremented the counter to 2

Thread 2 has released the lock.

Final value of shared_counter: 2

5. Program to implement Remote Method Invocation

```
#include <iostream>
#include <string>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
using namespace std;
const int PORT = 8080;
void performAddition(int client_socket) {
    int num1, num2, sum;
    // Read numbers from the client
    read(client_socket, &num1, sizeof(num1));
    read(client_socket, &num2, sizeof(num2));
    sum = num1 + num2;
    // Send the result back to the client
    write(client_socket, &sum, sizeof(sum));
}
int main() {
    int server_fd, client_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    // Create socket file descriptor
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        cerr << "Socket creation failed!" << endl;
        return -1;
    }
    // Define the server address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
```

```
    address.sin_port = htons(PORT);
    // Bind the socket to the port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        cerr << "Binding failed!" << endl;
        return -1;
    }
    // Start listening for connections
    listen(server_fd, 3);
    cout << "Server is listening on port " << PORT << endl;
    // Accept a client connection
    client_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen);
    if (client_socket < 0) {
        cerr << "Connection failed!" << endl;
        return -1;
    }
    cout << "Client connected!" << endl;
    // Perform addition
    performAddition(client_socket);
    // Close the socket
    close(client_socket);
    close(server_fd);
    return 0;
}
```

Client Code (client.cpp)

```
#include <iostream>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
using namespace std;
const int PORT = 8080;
int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        cerr << "Socket creation error!" << endl;
        return -1;
    }
    // Define the server address
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        cerr << "Invalid address/Address not supported!" << endl;
        return -1;
    }
    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        cerr << "Connection failed!" << endl;
        return -1;
    }
    // Input numbers to add
    int num1, num2;
```

```
    cout << "Enter two numbers to add: ";
    cin >> num1 >> num2;
    // Send numbers to the server
    write(sock, &num1, sizeof(num1));
    write(sock, &num2, sizeof(num2));
    // Receive the result from the server
    int sum;
    read(sock, &sum, sizeof(sum));
    cout << "Result from server: " << sum << endl;
    // Close the socket
    close(sock);
    return 0;
}
```

Sample Output

On the server terminal:

arduino

Server is listening on port 8080

Client connected!

On the client terminal:

sql

Enter two numbers to add: 5 7

Result from server: 12

6. Program to implement Remote Procedure Call.

Server Code (server.cpp)

```
#include <iostream>
#include <string>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
using namespace std;
const int PORT = 8080;
void performAddition(int client_socket) {
   int num1, num2, sum;
   // Read numbers from the client
   read(client_socket, &num1, sizeof(num1));
   read(client_socket, &num2, sizeof(num2));
   sum = num1 + num2;
   // Send the result back to the client
   write(client_socket, &sum, sizeof(sum));
}
int main() {
   int server_fd, client_socket;
   struct sockaddr_in address;
   int addrlen = sizeof(address);
   // Create socket file descriptor
   server_fd = socket(AF_INET, SOCK_STREAM, 0);
   if (server_fd == 0) {
      cerr << "Socket creation failed!" << endl;
      return -1;
   }
   // Define the server address
   address.sin_family = AF_INET;
```

```
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    // Bind the socket to the port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        cerr << "Binding failed!" << endl;
        return -1;
    }
    // Start listening for connections
    listen(server_fd, 3);
    cout << "Server is listening on port " << PORT << endl;
    // Accept a client connection
    client_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen);
    if (client_socket < 0) {
        cerr << "Connection failed!" << endl;
        return -1;
    }
    cout << "Client connected!" << endl;
    // Perform addition
    performAddition(client_socket);
    // Close the socket
    close(client_socket);
    close(server_fd);
    return 0;
}
```

Client Code (client.cpp)

```
#include <iostream>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
using namespace std;
const int PORT = 8080;
int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        cerr << "Socket creation error!" << endl;
        return -1;
    }
    // Define the server address
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        cerr << "Invalid address/Address not supported!" << endl;
        return -1;
    }
    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        cerr << "Connection failed!" << endl;
        return -1;
    }
    // Input numbers to add
    int num1, num2;
```

```
    cout << "Enter two numbers to add: ";
    cin >> num1 >> num2;
    // Send numbers to the server
    write(sock, &num1, sizeof(num1));
    write(sock, &num2, sizeof(num2));
    // Receive the result from the server
    int sum;
    read(sock, &sum, sizeof(sum));
    cout << "Result from server: " << sum << endl;
    // Close the socket
    close(sock);
    return 0;
}
```

Sample Output

On the server terminal:

arduino

Server is listening on port 8080

Client connected!

On the client terminal:

sql

Enter two numbers to add: 5 7

Result from server: 12

7.Program to implement Chat Server

Chat Server Implementation (server.cpp)

```
#include <iostream>
#include <string>
#include <vector>
#include <thread>
#include <mutex>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
using namespace std;
const int PORT = 8080;
vector<int> clients; // Vector to hold client sockets
mutex clientsMutex;  // Mutex to protect access to clients vector
void broadcastMessage(const string& message, int senderSocket) {
    lock_guard<mutex> lock(clientsMutex);
    for (int client : clients) {
        if (client != senderSocket) {
            send(client, message.c_str(), message.size(), 0);
        }
    }
}
void handleClient(int clientSocket) {
    char buffer[1024];
    string welcomeMessage = "Welcome to the chat server!\n";
    send(clientSocket, welcomeMessage.c_str(), welcomeMessage.size(), 0);
    while (true) {
```

```
        memset(buffer, 0, sizeof(buffer));
        int bytesRead = recv(clientSocket, buffer, sizeof(buffer), 0);
        if (bytesRead <= 0) {
            break; // Client disconnected
        }
        string message(buffer);
        cout << "Received: " << message;
        broadcastMessage(message, clientSocket);
    }
    // Remove client from the list and close the socket
    {
        lock_guard<mutex> lock(clientsMutex);
        clients.erase(remove(clients.begin(), clients.end(), clientSocket), clients.end());
    }
    close(clientSocket);
}
int main() {
    int serverSocket, clientSocket;
    struct sockaddr_in serverAddr, clientAddr;
    socklen_t clientAddrLen = sizeof(clientAddr);
    // Create socket
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == -1) {
        cerr << "Socket creation failed!" << endl;
        return -1;
    }
    // Define server address
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
```

```
    serverAddr.sin_port = htons(PORT);
    // Bind the socket to the port
    if (bind(serverSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0) {
        cerr << "Binding failed!" << endl;
        return -1;
    }
    // Start listening for connections
    listen(serverSocket, 5);
    cout << "Chat server is listening on port " << PORT << endl;
    while (true) {
        // Accept a new client connection
        clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddr, &clientAddrLen);
        if (clientSocket < 0) {
            cerr << "Connection failed!" << endl;
            continue;
        }
        {
            lock_guard<mutex> lock(clientsMutex);
            clients.push_back(clientSocket);
        }
        cout << "New client connected!" << endl;
        // Handle client in a new thread
        thread(handleClient, clientSocket).detach();
    }
    close(serverSocket);
    return 0;
}
```

Chat Client Implementation (client.cpp)

```
#include <iostream>
#include <thread>
#include <string>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
using namespace std;
const int PORT = 8080;
void receiveMessages(int socket) {
    char buffer[1024];
    while (true) {
        memset(buffer, 0, sizeof(buffer));
        int bytesRead = recv(socket, buffer, sizeof(buffer), 0);
        if (bytesRead <= 0) {
            break; // Server disconnected
        }
        cout << "Message: " << buffer << endl;
    }
}
int main() {
    int socket_fd;
    struct sockaddr_in serverAddr;

    // Create socket
    socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
```

```
        cerr << "Socket creation error!" << endl;
        return -1;
    }
    // Define the server address
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, "127.0.0.1", &serverAddr.sin_addr) <= 0) {
        cerr << "Invalid address/Address not supported!" << endl;
        return -1;
    }
    // Connect to the server
    if (connect(socket_fd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0) {
        cerr << "Connection to server failed!" << endl;
        return -1;
    }
    // Start a thread to receive messages
    thread(receiveMessages, socket_fd).detach();
    // Main loop to send messages
    string message;
    while (true) {
        cout << "Enter your message: ";
        getline(cin, message);
        send(socket_fd, message.c_str(), message.size(), 0);
    }
    close(socket_fd);
    return 0;
}
```

Sample Output

On the server terminal:

vbnet

Chat server is listening on port 8080

New client connected!

New client connected!

Received: Hello from client 1

Message: Hello from client 1

On the client terminal:

mathematica

Enter your message: Hello from client 1

Message: Hello from client 1

Enter your message: Hi there!

8. Program to implement termination detection

Process Class Definition

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>
#include <queue>
using namespace std;
class Process {
public:
    Process(int id) : id(id), state(0), terminationDetected(false) {}
    void run();
    void sendMessage(Process &receiver, const string &message);
    void receiveMessage(const string &message);
    void setPassive();
    void checkTermination();
    int id;
    int state; // 0 for active, 1 for passive
    bool terminationDetected;
private:
    mutex mtx;
    condition_variable cv;
    queue<string> messageQueue;
};
```

```
void Process::sendMessage(Process &receiver, const string &message) {
    unique_lock<mutex> lock(receiver.mtx);
    receiver.messageQueue.push(message);
    cv.notify_all(); // Notify the receiving process
}
void Process::receiveMessage(const string &message) {
    unique_lock<mutex> lock(mtx);
    messageQueue.push(message);
    cout << "Process " << id << " received message: " << message << endl;
}
void Process::setPassive() {
    unique_lock<mutex> lock(mtx);
    state = 1; // Set to passive
    cout << "Process " << id << " is now passive." << endl;
    cv.notify_all(); // Notify termination checking
}
void Process::checkTermination() {
    unique_lock<mutex> lock(mtx);
    if (state == 1 && messageQueue.empty()) {
        terminationDetected = true;
        cout << "Termination detected in process " << id << "!" << endl;
    }
}
```

Main Function to Simulate the System

```
int main() {
    Process p1(1);
    Process p2(2);
    // Simulating sending messages
    thread t1([&]() {
        this_thread::sleep_for(chrono::seconds(1));
        p1.sendMessage(p2, "Hello from Process 1");
        this_thread::sleep_for(chrono::seconds(1));
        p1.setPassive();
    });
    thread t2([&]() {
        this_thread::sleep_for(chrono::seconds(2));
        p2.sendMessage(p1, "Hello from Process 2");
        this_thread::sleep_for(chrono::seconds(1));
        p2.setPassive();
    });
    // Simulating message receiving and termination checking
    while (!p1.terminationDetected || !p2.terminationDetected) {
        this_thread::sleep_for(chrono::milliseconds(500));
        p1.checkTermination();
        p2.checkTermination();
        // Check for incoming messages
        if (!p1.messageQueue.empty()) {
            string msg = p1.messageQueue.front();
```

```
            p1.messageQueue.pop();
            p2.receiveMessage(msg);
        }
        if (!p2.messageQueue.empty()) {
            string msg = p2.messageQueue.front();
            p2.messageQueue.pop();
            p1.receiveMessage(msg);
        }
    }
    t1.join();
    t2.join();
    return 0;
}
```

Sample Output

Process 1 received message: Hello from Process 2

Process 2 received message: Hello from Process 1

Process 1 is now passive.

Process 2 is now passive.

Termination detected in process 1!

Termination detected in process 2!