NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

DISTRIBUTED COMPUTING

CACSC15

Mrs. Raina Joon

Sneha Gupta

2021UCA1859

Assignment 1

Program to implement non token based algorithm for Mutual Exclusion

Theory

In concurrent programming, mutual exclusion is a fundamental concept that ensures that only one thread or process accesses a shared resource at any given time. This is essential to prevent race conditions and maintain data consistency in multi-threaded or multi-process applications.

```cpp
#include <iostream>
#include <vector>
#include <unistd.h>

using namespace std;

// Define a struct to represent a request message
struct RequestMessage {
 int processId;
};

// Define a struct to represent a reply message
struct ReplyMessage {
 int processId;
};

// Define a class to represent a process
class Process {
private:
 int id;
 vector<RequestMessage> pendingRequests;
 vector<ReplyMessage> receivedReplies;
 bool inCriticalSection = false;

public:
 Process(int id) {
  this->id = id;
 }

  // Declare the processes vector inside the Process class
```

```cpp
public:
 Process(int id) {
  this->id = id;
 }

  // Declare the processes vector inside the Process
  class
  static vector<Process> processes;

 // Send a request message to all other processes
 void sendRequest() {
  RequestMessage requestMessage;
  requestMessage.processId = id;

  for (int i = 0; i < processes.size(); i++) {
   if (i != id) {
    processes[i].receiveRequest(requestMessage);
   }
  }
 }

 // Receive a request message from another process
 void receiveRequest(RequestMessage requestMessage) {
  pendingRequests.push_back(requestMessage);

  if (!inCriticalSection) {
   sendReply(requestMessage.processId);
```

```cpp
// Send a reply message to the requesting process
void sendReply(int requestingProcessId) {
 ReplyMessage replyMessage;
 replyMessage.processId = id;

 processes[requestingProcessId].receiveReply
 (replyMessage);
}

// Receive a reply message from another process
void receiveReply(ReplyMessage replyMessage) {
 receivedReplies.push_back(replyMessage);

 if (canEnterCriticalSection()) {
  enterCriticalSection();
 }
}

// Check if the process can enter the critical section
bool canEnterCriticalSection() {
 return receivedReplies.size() == processes.size() - 1;
}

// Enter the critical section
void enterCriticalSection() {
 cout << "Process " << id << " is entering the critical
 section" << endl;
```

```cpp
 // Simulate critical section execution
 sleep(1);

 cout << "Process " << id << " is leaving the critical
 section" << endl;

 inCriticalSection = false;
 pendingRequests.clear();
 receivedReplies.clear();
 }
};

// Define the processes vector outside of the Process
class
vector<Process> Process::processes;

int main() {
 // Create three processes

 Process::processes.push_back(Process(0));
 Process::processes.push_back(Process(1));
 Process::processes.push_back(Process(2));

 // Let each process try to enter the critical section
 for (int i = 0; i < Process::processes.size(); i++) {
  Process::processes[i].sendRequest();
 }
```

```cpp
 // Let each process try to enter the critical section
 for (int i = 0; i < Process::processes.size(); i++) {
  Process::processes[i].sendRequest();
 }

 while (true) {
  for (int i = 0; i < Process::processes.size(); i++) {
   if (Process::processes[i].canEnterCriticalSection()) {
    Process::processes[i].enterCriticalSection();
   }
  }
 }

 return 0;
}
```

OUTPUT:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS
                                   ~~~~~~~~
                                   Process
[Done] exited with code=1 in 0.488 seconds

[Running] cd "/home/asmo/Downloads/Distributed Computing Lab Files/" && g++ first.cpp -o first && "/home/asmo/Downloads/Distributed Computing Lab Files/"first
Process 0 is entering the critical section
Process 0 is leaving the critical section
Process 1 is entering the critical section
Process 1 is leaving the critical section
Process 2 is entering the critical section
Process 2 is leaving the critical section
```

Assignment 2

Program to implement Lamport’s Logical Clock

Theory:

Lamport's Logical Clock is a simple algorithm for ordering events in a distributed system. It does not rely on physical time but rather assigns a logical timestamp to each event. The key idea is that if event A happened before event B, their logical timestamps should reflect this relationship.

The algorithm works as follows:

1. Each process maintains a local logical clock, initially set to zero.

2. When a process performs an event (e.g., sending a message or receiving a message), it increments its logical clock by one and timestamps the event with the current value of its logical clock.

3. When a process receives a message with a timestamp, it updates its local logical clock to be the maximum of its current value and the timestamp received in the message, plus one. It then timestamps the event with this new logical clock value.

4. Events can be compared based on their logical timestamps. If Event A has a lower logical timestamp than Event B, it means A happened before B.

```
#include <iostream>

using namespace std;

class Process {
private:
  int id;
  int logicalClock;

public:
  Process(int id) {
    this->id = id;
    this->logicalClock = 0;
  }

  void incrementLogicalClock() {
    logicalClock++;
  }

  void sendMessage(Process& receiver, int eventId) {
    logicalClock++;
    cout << "Process " << id << " sending message with event id " << eventId << " and
    logical clock " << logicalClock << endl;
    receiver.receiveMessage(*this, eventId);
  }

  void receiveMessage(Process& sender, int eventId) {
    logicalClock = max(logicalClock, sender.logicalClock) + 1;
    cout << "Process " << id << " received message with event id " << eventId << " and
    updated logical clock to " << logicalClock << endl;
  }
};

int main() {
  Process p1(1);
  Process p2(2);

  p1.sendMessage(p2, 1);
  p2.sendMessage(p1, 2);
  p1.sendMessage(p2, 3);

  return 0;
```

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    COMMENTS

cd "/home/asmo/Downloads/Distributed Computing Lab Files/" && g++ second.cpp -o second && "/home/asmo/Downloads/Distributed Computing Lab Files/"second
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$ cd "/home/asmo/Downloads/Distributed Computing Lab Files/" && g++ second.cpp -o second && "/home/asmo/Downloads/D
puting Lab Files/"second
Process 1 sending message with event id 1 and logical clock 1
Process 2 received message with event id 1 and updated logical clock to 2
Process 2 sending message with event id 2 and logical clock 3
Process 1 received message with event id 2 and updated logical clock to 4
Process 1 sending message with event id 3 and logical clock 5
Process 2 received message with event id 3 and updated logical clock to 6
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$
```

Assignment 3

Program to implement edge chasing distributed deadlock detection algorithm.

Theory: Lamport's Logical Clock is a simple algorithm for ordering events in a distributed system. It does not rely on physical time but rather assigns a logical timestamp to each event. The key idea is that if event A happened before event B, their logical timestamps should reflect this relationship.

The algorithm works as follows:

1. Each process maintains a local logical clock, initially set to zero.

2. When a process performs an event (e.g., sending a message or receiving a message), it increments its logical clock by one and timestamps the event with the current value of its logical clock.

3. When a process receives a message with a timestamp, it updates its local logical clock to be the maximum of its current value and the timestamp received in the message, plus one. It then timestamps the event with this new logical clock value.

4. Events can be compared based on their logical timestamps. If Event A has a lower logical timestamp than Event B, it means A happened before B.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Process {
  int id;
  int waitingFor;
};

vector<Process> processes;

void detectDeadlock(int initiator) {
  int currentProcess = initiator;
  int nextProcess = processes[currentProcess].waitingFor;

  while (nextProcess != initiator) {
    currentProcess = nextProcess;
    nextProcess = processes[currentProcess].waitingFor;
  }

  if (nextProcess == initiator) {
    cout << "Deadlock detected!" << endl;
  } else {
    cout << "No deadlock detected" << endl;
  }
}

int main() {
  // Create three processes and their waiting dependencies
  processes.push_back({0, 2});
  processes.push_back({1, 0});
  processes.push_back({2, 1});

  // Initiate deadlock detection from process 0
  detectDeadlock(0);

  return 0;
}
```

```
Process 2 received message with event id 3 and updated logical clock to 6
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$ cd "/home/asmo/Downloads/Distributed Computing Lab Files/" && g++ third.cpp -o third && "/home/asmo/Downloads/Distributed
ting Lab Files/"third
Deadlock detected!
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$
```

Assignment 4

Program to implement locking algorithm.

Theory:

A locking algorithm ensures that only one thread can access a critical section of code at a time, preventing data races and conflicts that may lead to incorrect results or program crashes. Locking mechanisms come in various forms, with mutex locks being one of the most common. Here's how they work:

Mutex (Mutual Exclusion): A mutex is a synchronization primitive that allows threads to lock access to a shared resource. When a thread locks a mutex, it enters a critical section. If another thread tries to lock the same mutex while it's locked by another thread, it will block until the first thread releases the lock.

Lock and Unlock: Threads must explicitly lock the mutex before entering the critical section and unlock it when they exit. This ensures mutual exclusion and prevents multiple threads from accessing the critical section simultaneously.

```cpp
#include <iostream>
#include <mutex>
#include <thread>

using namespace std;

mutex mtx; // Global mutex object

void criticalSection() {
  // Acquire the lock before entering the critical section
  mtx.lock();

  // Simulate critical section execution
  cout << "Thread " << this_thread::get_id() << " is in the
critical section" << endl;

  // Release the lock after exiting the critical section
  mtx.unlock();
}

int main() {
  thread t1(criticalSection);
  thread t2(criticalSection);

  t1.join();
  t2.join();

  return 0;
}
```

```
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$ cd "/home/asmo/Downloads/Distributed Computing Lab Files/" && g++ fourth.cpp -o fourth && "/home/asmo/Downloads/Distribute
puting Lab Files/"fourth
Thread 139621648299584 is in the critical section
Thread 139621639906880 is in the critical section
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$
```

Assignment 5

Program to implement Remote Method Invocation.

Theory:

Remote Method Invocation (RMI) is a mechanism that allows an object in one address space (usually on a remote machine) to invoke methods on an object in another address space, possibly on a different machine. RMI is a fundamental concept in distributed computing and is typically used in client-server applications. It enables communication between objects in different address spaces by making method calls appear as if they were local.

```cpp
#include <iostream>
#include <string>

using namespace std;

class RemoteObject {
public:
    virtual string remoteMethod() = 0;
};

class LocalObject : public RemoteObject {
public:
    string remoteMethod() override {
        return "Hello from the local object!";
    }
};

class Client {
public:
    string callRemoteMethod(RemoteObject* remoteObj) {
        return remoteObj->remoteMethod();
    }
};

int main() {
    LocalObject localObject;
    Client client;

    RemoteObject* remoteObj = &localObject; // Simulate a remote
    object

    // Invoke the remote method
    string result = client.callRemoteMethod(remoteObj);

    cout << "Result from the remote method: " << result << endl;

    return 0;
}
```

```
Result from the remote method: Hello from the local object!
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$ cd "/home/asmo/Downloads/Distributed Computing Lab Files/" && g++ fifth.cpp -o fifth && "/home/asmo/Downloads/Distributed
ting Lab Files/"fifth
Result from the remote method: Hello from the local object!
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$
```

Assignment 6

Program to implement Remote Procedure Call. Remote Procedure Call (RPC) is a powerful mechanism for invoking procedures or methods in a different address space, typically on a remote machine. RPC allows distributed systems to make a local procedure call on a remote system as if it were a local call. In C++, this is often implemented using technologies such as gRPC or Apache Thrift.

Theory:

RPC involves two main components:

1. Client: The client initiates a procedure call, and the local stub (proxy) prepares a request message. This message is then sent to the server using a network protocol. Distributed Computing 11

2. Server: The server receives the request message and forwards it to the local stub (skeleton). The local stub calls the actual procedure/method on the server. The result is then sent back to the client in a response message.

```cpp
#include <iostream>
#include <string>

using namespace std;

// Define a remote interface
class RemoteService {
public:
    virtual int add(int a, int b) = 0;
};

// Server implementation
class RemoteServiceImpl : public RemoteService {
public:
    int add(int a, int b) override {
        return a + b;
    }
};

// Client implementation
class RPCClient {
public:
    int callRemoteAdd(int a, int b, RemoteService* service) {
        return service->add(a, b);
    }
};

int main() {
    // Simulate server and client
    RemoteServiceImpl server;
    RPCClient client;

    // The client invokes a remote procedure
    int result = client.callRemoteAdd(5, 3, &server);

    cout << "Result of the remote procedure call: " << result <<
    endl;

    return 0;
}
```

```
Result of the remote procedure call: 8
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$ cd "/home/asmo/Downloads/Distributed Computing Lab Files/" && g++ sixth.cpp -o sixth && "/home/asmo/Downloads/Distributed
ting Lab Files/"sixth
Result of the remote procedure call: 8
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$
```

Assignment 7

Program to implement Chat Server

```cpp
#include <iostream>
#include <vector>

using namespace std;

class Client {
private:
    string name;

public:
    Client(string name) {
        this->name = name;
    }

    string getName() {
        return name;
    }
};

vector<Client> clients;

void broadcastMessage(string message) {
    for (Client& client : clients) {
        cout << "Sending message to " << client.getName() << ": " << message << endl;
    }
}

int main() {
    // Add some clients to the chat
    clients.push_back(Client("Alice"));
    clients.push_back(Client("Bob"));
    clients.push_back(Client("Charlie"));

    // Simulate receiving messages from clients and broadcasting them
    broadcastMessage("Hello from Alice!");
    broadcastMessage("Hi there, from Bob!");
    broadcastMessage("Greetings everyone, Charlie here!");

    return 0;
}
```

```
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$ cd "/home/asmo/Downloads/Distributed Computing Lab Files/" && g++ seventh.cpp -o seventh && "/home/asmo/Downloads/Distrib
omputing Lab Files/"seventh
Sending message to Alice: Hello from Alice!
Sending message to Bob: Hello from Alice!
Sending message to Charlie: Hello from Alice!
Sending message to Alice: Hi there, from Bob!
Sending message to Bob: Hi there, from Bob!
Sending message to Charlie: Hi there, from Bob!
Sending message to Alice: Greetings everyone, Charlie here!
Sending message to Bob: Greetings everyone, Charlie here!
Sending message to Charlie: Greetings everyone, Charlie here!
asmo@Asmo:~/Downloads/Distributed Computing Lab Files$
```

Assignment 8

Program to implement termination detection

Termination detection is an essential concept in distributed systems, ensuring that the system can determine when a distributed computation or set of processes has finished. Termination detection can be useful for various purposes, such as resource reclamation, signaling completion, or initiating another phase of a distributed computation.

```cpp
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>

using namespace std;

class Process {
private:
    int id;
    bool isFinished;
    mutex mtx;

public:
    Process(int id) {
        this->id = id;
        this->isFinished = false;
    }

    void run() {
        // Simulate process execution

        // Set the finished flag to true
        mtx.lock();
        isFinished = true;
        mtx.unlock();
    }

    bool isFinished() {
        mtx.lock();
        bool finished = isFinished;
        mtx.unlock();

        return finished;
    }
};

vector<Process> processes;

void terminate() {
    cout << "All processes have finished executing" << endl;
}
```

```cpp
void monitorTermination() {
    while (true) {
        // Check if all processes have finished executing
        bool allFinished = true;
        for (Process& process : processes) {
            if (!process.isFinished()) {
                allFinished = false;
                break;
            }
        }

        // If all processes have finished, terminate the system
        if (allFinished) {
            terminate();
            break;
        }

        // Wait for a short period before checking again
        this_thread::sleep_for(chrono::milliseconds(100));
    }
}

int main() {
    // Create a few processes
    processes.push_back(Process(0));
    processes.push_back(Process(1));
    processes.push_back(Process(2));

    // Start the processes
    for (Process& process : processes) {
        thread t(&Process::run, &process);
        t.detach();
    }

    // Start the termination monitor thread
    thread monitor(monitorTermination);

    monitor.join();

    return 0;
}
```

Assignment 9

To implement CORBA mechanism by using C++ program at one end and Java Program on the other.

Theory:

CORBA (Common Object Request Broker Architecture) is a standard for distributed object computing that allows objects on different machines to communicate with each other as if they were local. CORBA uses an IDL (Interface Definition Language) to define the interfaces of distributed objects, which can then be implemented in different programming languages.

```
#include <iostream>
#include <CORBA.h>

using namespace std;

// Define the IDL interface for the distributed object
class HelloService {
public:
  virtual string sayHello(string name) = 0;
};

// Implement the CORBA servant
class HelloServiceImpl : public virtual CORBA::Object, public
HelloService {
public:
  HelloServiceImpl() {}

  string sayHello(string name) override {
    return "Hello, " + name + "!";
  }
};

int main() {
  // Initialize the ORB
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

  // Create the servant
  HelloServiceImpl* servant = new HelloServiceImpl();

  // Register the servant with the ORB
  CORBA::Object_var objectRef = orb->register_initial_reference
  ("HelloService", servant);

  // Start the ORB and wait for requests
  orb->run();

  return 0;
}
```