

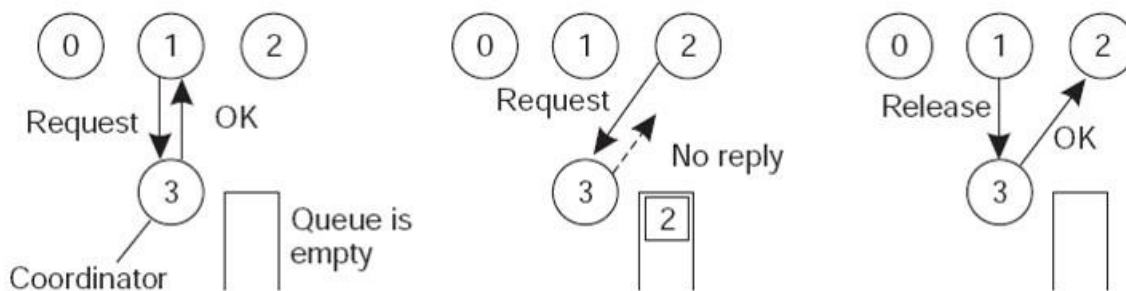
PROGRAM – 4

Aim: Program to implement Mutual Exclusion using centralized algorithm.

Theory:

In centralized algorithm one process is elected as the coordinator which may be the machine. Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission). When the reply arrives, the requesting process enters the critical region. When another process asks for permission to enter the same critical region. Now the coordinator knows that a different process is already in the critical region, so it cannot grant permission.

The coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply or it could send a reply 'permission denied.' When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access. The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. When it sees the grant, it can enter the critical region



Algorithm:

Coordinator Loop

```
receive(msg);  
case msg of REQUEST:  
    if nobody in CS  
        then reply GRANTED  
    else queue the REQ;  
        reply DENIED  
RELEASE:  
    if queue not empty  
        then remove 1st on the queue  
            reply GRANTED  
end case
```

end loop

Client:

send(REQUEST);

receive(msg);

if msg != GRANTED

then receive(msg);

enter CS;

send(RELEASE)

Code:

Server:

```
import java.io.*;
```

```
import java.net.*;
```

```
public class Server implements Runnable {
    Socket socket = null;
    static ServerSocket ss;
    Server(Socket newSocket) {
        this.socket = newSocket;
    }

    public static void main(String args[]) throws IOException {
        ss = new ServerSocket(7000);
        System.out.println("Server Started");

        while (true) {
            Socket s = ss.accept();
            Server es = new Server(s);
            Thread t = new Thread(es);
            t.start();
        }
    }

    public void run() {
        try {
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            while (true) {
                System.out.println(in.readLine());
            }
        } catch (Exception e) {
        }
    }
}
```

Client1:

```
import java.io.*;
import java.net.*;

public class Client1 {
    public static void main(String args[]) throws IOException {
        Socket s = new Socket("localhost", 7000);
        PrintStream out = new PrintStream(s.getOutputStream());

        ServerSocket ss = new ServerSocket(7001);
        Socket s1 = ss.accept();
        BufferedReader in1 = new BufferedReader(new
InputStreamReader(s1.getInputStream()));
        PrintStream out1 = new PrintStream(s1.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str = "Token";
        while (true) {
            if (str.equalsIgnoreCase("Token")) {
                System.out.println("Do you want to send some data");
                System.out.println("Enter Yes or No");
                str = br.readLine();
                if (str.equalsIgnoreCase("Yes")) {
                    System.out.println("Enter the data");
                    str = br.readLine();
                    out.println(str);
                }
                out1.println("Token");
            }
            System.out.println("Waiting for Token");
            str = in1.readLine();
        }
    }
}
```

Client2:

```
import java.io.*;
import java.net.*;

public class Client2 {
    public static void main(String args[]) throws IOException {
        Socket s = new Socket("localhost", 7000);
        PrintStream out = new PrintStream(s.getOutputStream());

        Socket s2 = new Socket("localhost", 7001);
        BufferedReader in2 = new BufferedReader(new
InputStreamReader(s2.getInputStream()));
        PrintStream out2 = new PrintStream(s2.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        while (true) {
            System.out.println("Waiting for Token");
            str = in2.readLine();
```

```

        if (str.equalsIgnoreCase("Token")) {
            System.out.println("Do you want to send some data");
            System.out.println("Enter Yes or No");
            str = br.readLine();
            if (str.equalsIgnoreCase("Yes")) {
                System.out.println("Enter the data");
                str = br.readLine();
                out.println(str);
            }
            out2.println("Token");
        }
    }
}
}

```

Output:

Server:

```

1: java
mcsnipe97@NP515-Anuj:~/wsl/labs/4$ java Server
Server Started
Morning
Night
█

```

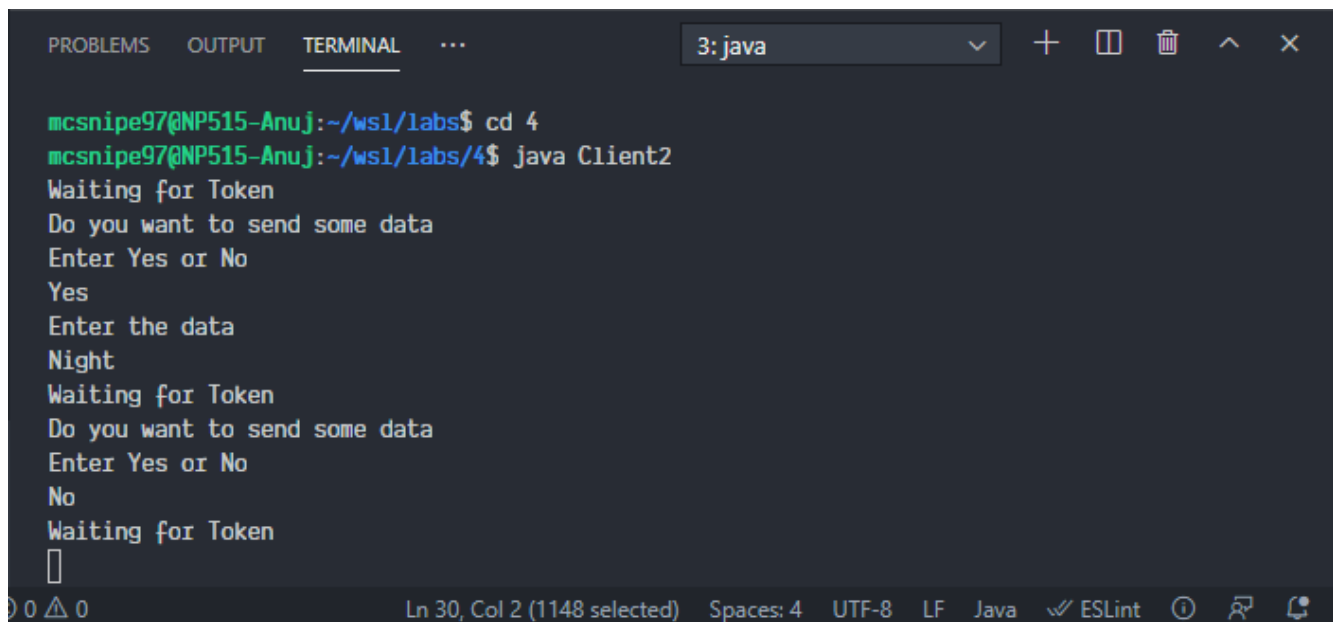
Client 1:

```

2: java
mcsnipe97@NP515-Anuj:~/wsl/labs/4$ java Client1
Do you want to send some data
Enter Yes or No
Yes
Enter the data
Morning
Waiting for Token
Do you want to send some data
Enter Yes or No
No
Waiting for Token
Do you want to send some data
Enter Yes or No
█

```

Client 2:



```
mcsnipe97@NP515-Anuj:~/wsl/labs$ cd 4
mcsnipe97@NP515-Anuj:~/wsl/labs/4$ java Client2
Waiting for Token
Do you want to send some data
Enter Yes or No
Yes
Enter the data
Night
Waiting for Token
Do you want to send some data
Enter Yes or No
No
Waiting for Token
[]
```

Conclusion:

In this experiment, we implemented mutual exclusion using a centralized server. It simulates centralized locking with blocking calls. It is a simple method with no starvation and is fair. Although, it has the disadvantage of single point failure and is a performance bottleneck. This algorithm guarantees mutual exclusion by letting one process at a time into each critical region. It is also fair as requests are granted in the order in which they are received. It is easy to implement since it requires only three messages per use of a critical region (request, grant, release). This algorithm is also used for more general resource allocation rather than just managing critical regions.