Experiment 5

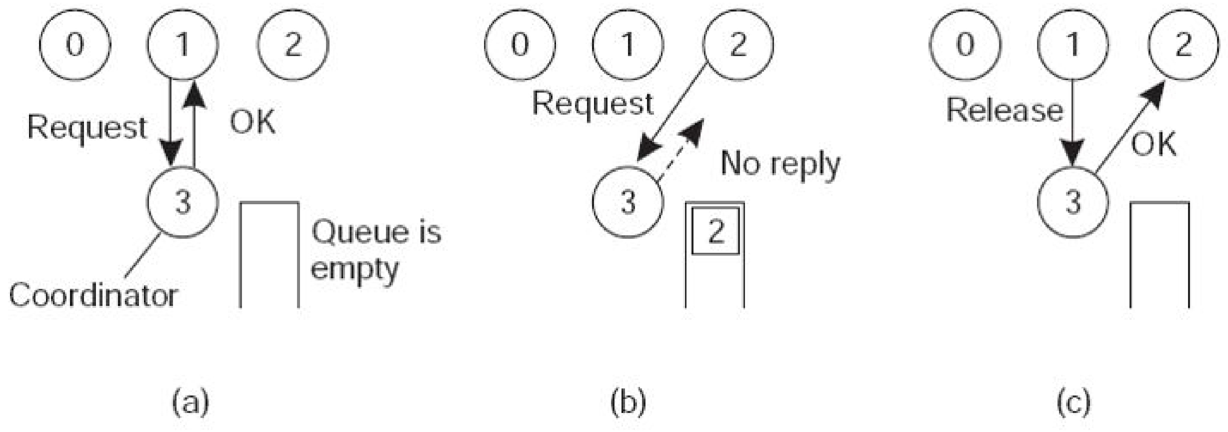
AIM: Write a program to implement Mutual Exclusion to access a shared file using centralized algorithm.

THEORY:

Mutual Exclusion: Processes in a distributed system may need to simultaneously access the same resource. Thus, there is a need to grant mutual exclusive access to shared resources by processes. This can be solved via a centralized server algorithm – Decentralized, using a peer‐to‐peer system – Distributed, with no topology imposed – Distributed, along a logical ring algorithm.

Centralized Algorithm

- In centralized algorithm one process is elected as the coordinator which may be the machine with the highest network address.
- 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, as shown in Fig (a). When the reply arrives, the requesting process enters the critical region



- Suppose another process 2 shown in Fig (b), 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 saying 'permission denied.'
- When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access as shown in Fig (c).

- 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 unlocks 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

Input - No. of Processes, Coordinator id, Resource id and request list.
Output - Sequence in which events are executed using Centralised mutual exclusion.

```
1. Start
2. Input No, of Processes, Coordinator id and Resource id.
3. Input Request list.
4. Print all the details.
5. For all processed in request list and while queue is not empty do:
    a. If resource is available:
        i. Allot the resource to requesting process and change resource status
    b. Else:
        i. Add process in queue and wait till resource is released and allot the
           resource to the first process in queue.
6. Print all sequences of events as they happen.
7. Stop
```

SOURCE CODE:

Controller Code

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

#define TRUE 1
#define FALSE 0
```

```
typedef struct resources
{
        int A;
        int B;
        int C;
        int D;
} resources;

int main()
{
        resources R, temp;
        R.A = 1;
        R.B = 2;
        R.C = 3;
        R.D = 4;
        FILE *fle;
        fle = fopen("shared_mem.txt", "w");
        fwrite(&R, sizeof(R), 1, fle);
        fclose(fle);
        struct sockaddr_in sa; // Socket address data structure
        int opt = TRUE, addrlen;
        int sockfd, clients[50]; // Source and destination addresses
        char buff[256];                         // Buffer to hold the out-going stream
        int rec, i, sd, activity, new_sock, sended;
        int max_sd;
        int flag = 0;
        sockfd = socket(AF_INET, SOCK_STREAM, 0); // New socket created
                                                // Checking for valid socket
        memset(clients, 0, sizeof(clients));

        fd_set readfds;
        if (sockfd < 0)
        {
                printf("Error in creating socket\n");
                exit(0);
        }
        else
        {
                printf("Socket Created\n");
        }
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (char *)&opt, sizeof(opt)) < 0)
        {
                printf("error\n");
        }
        // Clearing and assigning type and address to the socket
        printf("Socket created\n");
        bzero(&sa, sizeof(sa));
        sa.sin_family = AF_INET;
        sa.sin_port = htons(8888);
        sa.sin_addr.s_addr = htonl(INADDR_ANY);

        // binding and verifying the socket to address
        if (bind(sockfd, (struct sockaddr *)&sa, sizeof(sa)) < 0)
        {
                printf("Bind Error\n");
        }
        else
                printf("Binded\n");

        // starts the server with a max client queue size set as 10
        listen(sockfd, 10);
        addrlen = sizeof(sa);
        // server run
        while (TRUE)
        {
                // Clearing socket set
                FD_ZERO(&readfds);
                FD_SET(sockfd, &readfds);
                max_sd = sockfd;
                for (i = 0; i < 50; i++)
                {       sd = clients[i];
                        if (sd > 0)
                                FD_SET(sd, &readfds);
                        if (sd > max_sd)
                                max_sd = sd;
                }
                activity = select(max_sd + 1, &readfds, NULL, NULL, NULL);
                if (activity < 0)
```

```
				printf("Select error\n");
			if (FD_ISSET(sockfd, &readfds))
			{
				if ((new_sock = accept(sockfd, (struct sockaddr *)NULL, NULL)) < 0)
					perror("accept");
				else
				{	printf("New connection, sock fd %d\n", new_sock);
				}
				sended = send(new_sock, buff, strlen(buff), 0);
				if (sended < 0)
					perror("Send");
				for (i = 0; i < 50; i++)
				{
					if (clients[i] == 0)
					{
						clients[i] = new_sock;
						break;
					}
				}
			}
			for (i = 0; i < 50; i++)
			{	sd = clients[i];
				if (FD_ISSET(sd, &readfds))
				{
					FILE *fle;
					fle = fopen("shared_mem.txt", "r");
					fread(&temp, sizeof(temp), 1, fle);
					fclose(fle);
					rec = read(sd, buff, 256);
				    if (rec == 0)
				      {getpeername(sd, (struct sockaddr *)&sa, (socklen_t *)&sa);
					printf("%d has disconnected unexpectedly with ip %s and port
%d\n", sd, inet_ntoa(sa.sin_addr), ntohs(sa.sin_port));
					printf("recovering data\n");
						FILE *fle;
						fle = fopen("shared_mem.txt", "w+");
						fwrite(&temp, sizeof(temp), 1, fle);
						fclose(fle);
						close(sd);
```

```
                                        clients[i] = 0;
                                }
                                else
                                {
                                        buff[rec] = '\0';
                                        printf("recieved %s from %d\n", buff, sd);
                                        if (strcmp(buff, "PING") == 0 && flag == 1)
                                {
                                printf("Read buffer = %s, from %d and send NACK\n", buff, sd);
                                                sended = write(sd, "NACK", 4);
                                        }

                                        else if (strcmp(buff, "PING") == 0 && flag == 0)
                                        {
                                                printf("Read Buffer = %s, from %d\n", buff, sd);
                                                flag = 1;
                                                sended = write(sd, "PONG", 4);
                                        }
                                        else if (strcmp(buff, "DONE") == 0)
                                        {
                                                printf("Lock freed\n");
                                                flag = 0;
                                                FILE *fle;
                                                fle = fopen("shared_mem.txt", "r");
                                                fread(&temp, sizeof(temp), 1, fle);
                printf("Read %d, %d, %d, %d from %d\n", temp.A, temp.B, temp.C, temp.D, sd);
                                                fclose(fle);
                                                clients[i] = 0;
                                                close(sd);
                                                break;
                                        }
                                }
                        }
                }
        }
        close(sockfd); // close the socket
        return 0;
}
}
```

Process Code

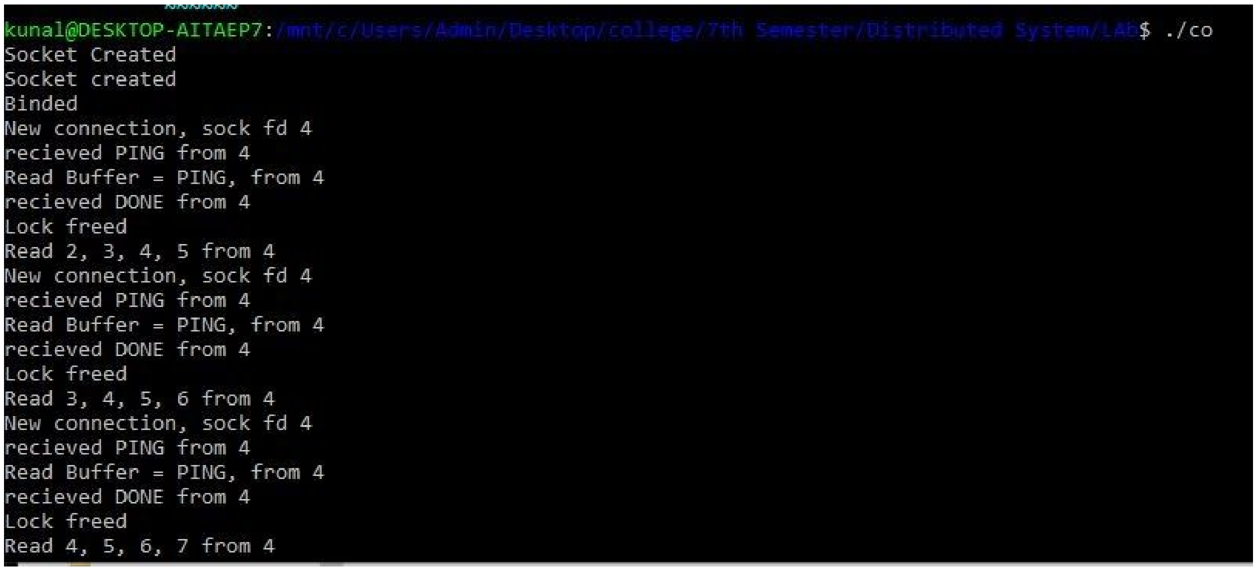
```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
#include<unistd.h>

typedef struct resources
{
   int A;
   char B;
   int C;
   char D;
}resources;

int main()
{
          struct sockaddr_in sa;  // Socket address data structure
   resources R;
          int n, sockfd; // read and source
          char buff[1025], obuff[256]; // buffer to store the read stream
          int snded, rec;

          sockfd = socket(PF_INET, SOCK_STREAM, 0); // New socket created


                                          // Checking for valid socket
          if (sockfd < 0)
          {
                    printf("Error in creation\n");
                    exit(0);
          }
          else
```

```
            printf("Socket created\n");

    // Clearing and assigning type and address to the socket
    bzero(&sa, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(8888);

    // establishing and verifying the connection
    if (connect(sockfd, (struct sockaddr_in*)&sa, sizeof(sa)) < 0)
    {
            printf("Connection failed\n");
            exit(0);
    }
    else
            printf("Connection made\n");

    while (1)
    {
            snded = write(sockfd, "PING", 5);
            if (snded > -1)
                    printf("SENT PING\n");
            rec = read(sockfd, obuff, 256);
            obuff[rec] = '\0';
            if (strcmp(obuff, "PONG") == 0)
  {
    usleep(750);
    FILE *f;
    f = fopen("shared_mem.txt", "r");
    fread(&R, sizeof(R), 1, f);
    fclose(f);
    printf("read %d, %d, %d, %d from server\n", R.A, R.B, R.C, R.D );
    R.A += 1;
    R.B += 1;
    R.C += 1;
    R.D += 1;
    f = fopen("shared_mem.txt", "w");
    fwrite(&R, sizeof(R), 1, f);
    fclose(f);
    printf("Got access to CS\n");
```

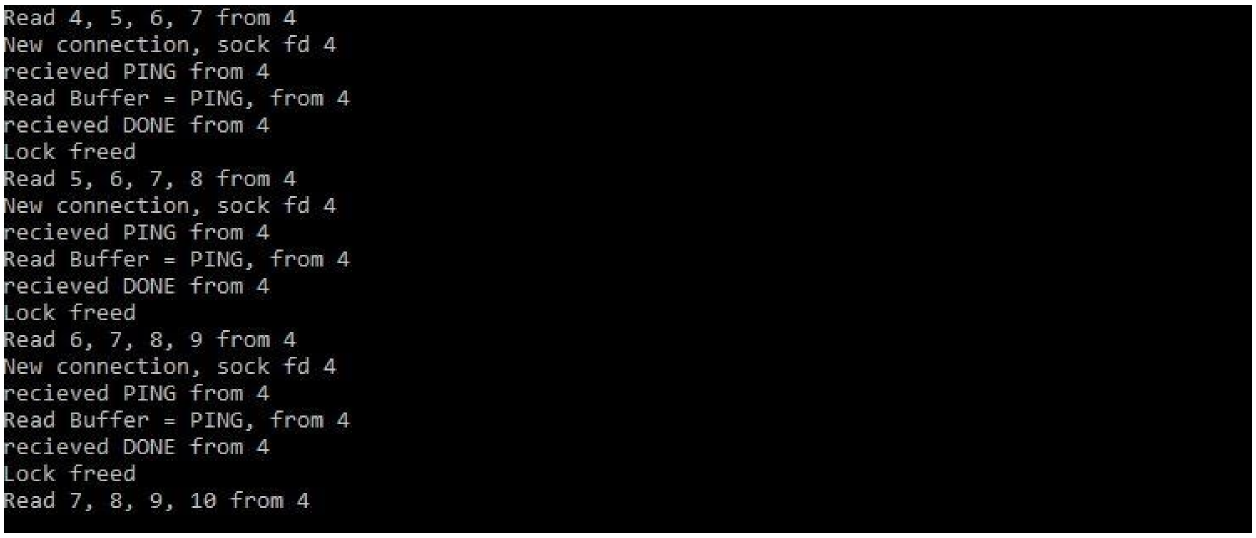
```
                        snded = write(sockfd, "DONE", 4);
            printf("Freeing Lock\n");
            break;
        }
            }
            // Reading and priting data from the server after verification
            close(sockfd); // Closing the socket
            return 0;
}
```

OUTPUT:

Controller

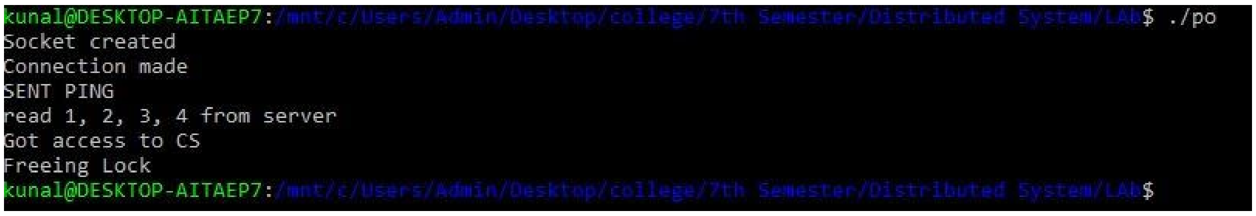


```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ ./co
Socket Created
Socket created
Binded
New connection, sock fd 4
recieved PING from 4
Read Buffer = PING, from 4
recieved DONE from 4
Lock freed
Read 2, 3, 4, 5 from 4
New connection, sock fd 4
recieved PING from 4
Read Buffer = PING, from 4
recieved DONE from 4
Lock freed
Read 3, 4, 5, 6 from 4
New connection, sock fd 4
recieved PING from 4
Read Buffer = PING, from 4
recieved DONE from 4
Lock freed
Read 4, 5, 6, 7 from 4
```



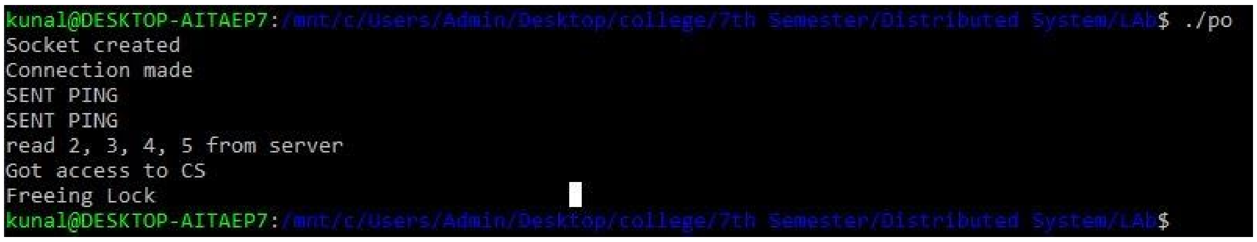
```
Read 4, 5, 6, 7 from 4
New connection, sock fd 4
recieved PING from 4
Read Buffer = PING, from 4
recieved DONE from 4
Lock freed
Read 5, 6, 7, 8 from 4
New connection, sock fd 4
recieved PING from 4
Read Buffer = PING, from 4
recieved DONE from 4
Lock freed
Read 6, 7, 8, 9 from 4
New connection, sock fd 4
recieved PING from 4
Read Buffer = PING, from 4
recieved DONE from 4
Lock freed
Read 7, 8, 9, 10 from 4
```

Process P1



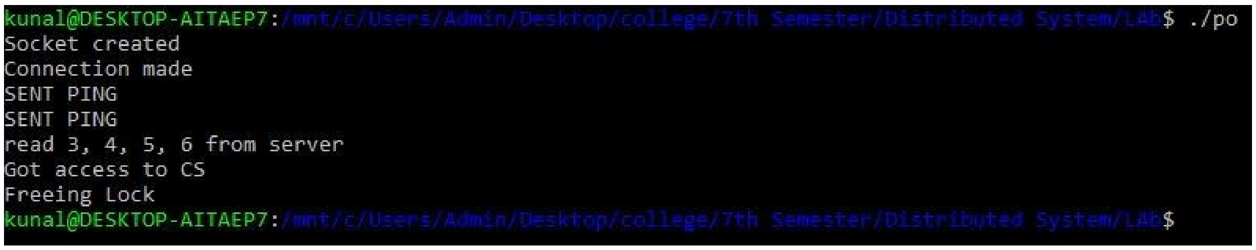
```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ ./po
Socket created
Connection made
SENT PING
read 1, 2, 3, 4 from server
Got access to CS
Freeing Lock
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$
```

Process P2



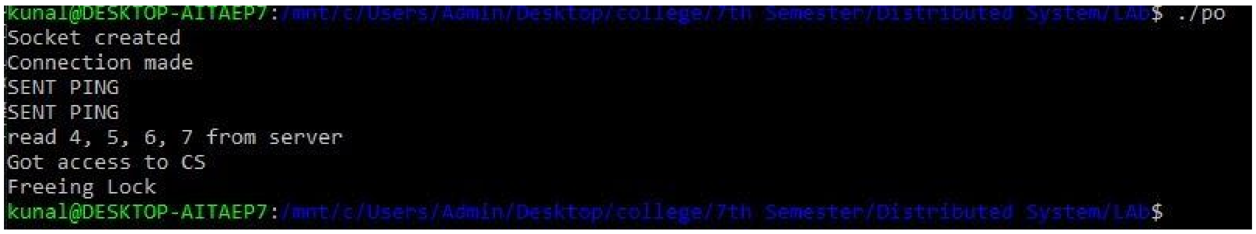
```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ ./po
Socket created
Connection made
SENT PING
SENT PING
read 2, 3, 4, 5 from server
Got access to CS
Freeing Lock
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$
```

Process P3



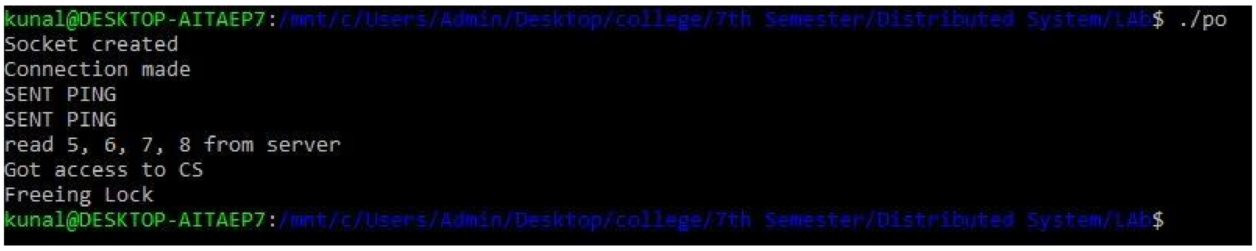
```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ ./po
Socket created
Connection made
SENT PING
SENT PING
read 3, 4, 5, 6 from server
Got access to CS
Freeing Lock
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$
```

Process P4



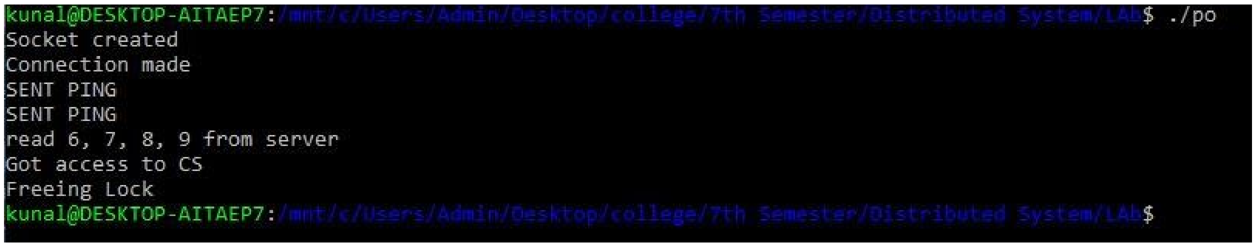
```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ ./po
Socket created
Connection made
SENT PING
SENT PING
read 4, 5, 6, 7 from server
Got access to CS
Freeing Lock
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$
```

Process P5



```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ ./po
Socket created
Connection made
SENT PING
SENT PING
read 5, 6, 7, 8 from server
Got access to CS
Freeing Lock
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$
```

Process P6



```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ ./po
Socket created
Connection made
SENT PING
SENT PING
read 6, 7, 8, 9 from server
Got access to CS
Freeing Lock
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$
```

LEARNING OUTCOMES:

The above program demonstrates how the problem of mutual exclusion is solved in a distributed system using centralised mutual exclusion algorithm. This is the most straightforward way to achieve mutual exclusion by simulating how it is done in a one-processor system.