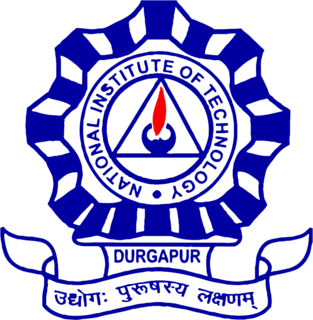
# Assignment **4** Of

**Network & Distributed System Lab (CS2051)**

**Masters of Technology in Computer Science And Engineering**

**submitted to**

**Dr Suvrojit Das Associate Professor Dept. of CSE**

****

**National Institute of Technology, Durgapur**

submitted by Arghya Bandyopadhyay RollNo. 20CS4103

01 July 2021

## **Objective:Write DS Code to implement Distributed Mutual Exclusion (DME). Consider a single file as a single resource and implement the classic Reader and Writers problem, considering writing to the file as a CS problem. Implement Lamport's solution for DME.**

## **Solution:**

//Implementation of Lamport logical clock for DME

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#define PORT 5000

#define FILENAME "sharedfile"

#define REQUEST\_SET

#define REQUEST\_MSG 100

#define REPLY\_MSG 200

#define RELEASE\_MSG 300

struct message {

short message\_type;

short message\_ts; short process\_id;

};

struct request\_queue {

short request\_ts;

short requesting\_process\_id;

struct request\_queue \*next\_request;

};

int err\_handler(int n, char \* error) {

if(n < 0) {

perror(error);

exit(EXIT\_FAILURE);

}

return n;

};

//This function creates socket and binds it to the port address specified

int \_create\_sock\_and\_bind(struct sockaddr\_in \* address, struct sockaddr\_in \*bcast\_address) {

int sock\_fd = err\_handler(socket(AF\_INET, SOCK\_DGRAM, 0), "SocketError");

int yes = 1;

err\_handler(setsockopt(sock\_fd, SOL\_SOCKET, SO\_BROADCAST,

&yes, sizeof(yes)), "BroadcastSettingError");

err\_handler(setsockopt(sock\_fd, SOL\_SOCKET, SO\_REUSEADDR,

&yes, sizeof(yes)), "ReuseAddrSettingError");

socklen\_t address\_size = sizeof(struct sockaddr\_in);

memset(address, 0, address\_size);

address->sin\_family = AF\_INET;

address->sin\_port = htons(PORT);

address->sin\_addr.s\_addr = INADDR\_ANY;

bcast\_address->sin\_family = AF\_INET;

bcast\_address->sin\_port = htons(PORT);

bcast\_address->sin\_addr.s\_addr = INADDR\_BROADCAST;

return sock\_fd;

}

//Function puts a request in the request queue

struct request\_queue \* \_put\_request\_in\_request\_queue(struct request\_queue \*queue, struct message \*msg) {

if(queue == NULL) {

queue = (struct request\_queue \*) malloc(sizeof(struct request\_queue));

queue->request\_ts = msg->message\_ts;

queue->requesting\_process\_id = msg->process\_id;

queue->next\_request = NULL;

}

else {

struct request\_queue \* temp = queue;

while(temp->next\_request != NULL) {

temp = temp->next\_request;

}

struct request\_queue \* new\_request = (struct request\_queue \*)

malloc(sizeof(struct request\_queue));

new\_request->request\_ts = msg->message\_ts;

new\_request->requesting\_process\_id = msg->process\_id;

new\_request->next\_request = NULL;

temp->next\_request = new\_request;

}

printf("Request added to request queue.\n");

return queue;

};

//This function sends request to write the shared file

struct request\_queue \* \_send\_cs\_request(int sock, short pid, struct message \*msg,

struct request\_queue \*queue, struct sockaddr\_in addr) {

printf("\nSending request, to write to the shared file, to all other processes\n");

msg->message\_type = REQUEST\_MSG;

msg->message\_ts += 1; /\* increment the timestamp for sending message \*/

msg->process\_id = pid;

queue = \_put\_request\_in\_request\_queue(queue, msg);

printf("Message timestamp: %d\n", msg->message\_ts);

err\_handler(sendto(sock, msg, sizeof(struct message), 0,

(struct sockaddr\_in \*) &addr, sizeof(struct sockaddr\_in)), "SendError");

return queue;

};

struct request\_queue \* \_recv\_cs\_request(int sock, struct request\_queue \*queue,

struct sockaddr\_in \*addr) {

printf("\nWaiting for some process's REQUEST to execute CS (write to the shared file)...\n");

struct message recvd\_msg; /\* to store the received msg \*/

memset(&recvd\_msg, 0, sizeof(struct message));

socklen\_t size\_addr = sizeof(struct sockaddr\_in);

err\_handler(recvfrom(sock, &recvd\_msg, sizeof(struct message), 0,

(struct sockaddr\_in \*) addr, &size\_addr), "RecvReqError");

printf("REQUEST Received.\n");

printf("Received timestamp: %d\n", recvd\_msg.message\_ts);

queue = \_put\_request\_in\_request\_queue(queue, &recvd\_msg);

return queue;

};

//This function receives the reply and makes sure that it is correct. If mot, it is ignored.

void \_recv\_reply(int sock, short pid, struct message \*msg, short \*perm) {

struct message recvd\_msg;

while(1) {

err\_handler(recvfrom(sock, &recvd\_msg, sizeof(struct message),

0, NULL, NULL), "ReceiveError");

if(recvd\_msg.message\_type == REPLY\_MSG && recvd\_msg.process\_id != pid) {

break;

}

}

printf("Received timestamp: %d\n", recvd\_msg.message\_ts);

if(recvd\_msg.message\_ts > msg->message\_ts) {

/\* the replying process is providing the permission to CS \*/

perm[recvd\_msg.process\_id - 1] = 1;

}

};

//This function checks the conditions of lamport clock.

int checkLamportCondition(struct request\_queue \*front, short perm[], short pid) {

for(int i = 0; i < REQUEST\_SET + 1; i++) {

if(perm[i] == 0) {

printf("process %d denied permission.\n", i+1);

/\* it means at least one permission from some process is missing\*/

return 0;

}

}

if(front->requesting\_process\_id != pid) {

printf("Process %d request is not at the front of request queue.\n", pid);

return 0;

}

printf("Conditions statisfied.\n");

printf("All processes have granted permission to process %d.\n", pid);

return 1;

};

void startWritingToFile(short pid) {

printf("\nEntering Critical Section. Writing to shared file...\n");

FILE \* shared\_file = fopen(FILENAME, "a");

int written;

if(shared\_file == NULL) {

perror("FileOpenError");

exit(EXIT\_FAILURE);

}

char line\_to\_write[40];

sprintf(line\_to\_write, "This an entry written by process %d\n", pid);

written = fputs(line\_to\_write, shared\_file);

if(written > 0) {

printf("\nProcess %d wrote %d bytes to the file.\n", pid, strlen(line\_to\_write));

}

else {

perror("WriteError");

exit(EXIT\_FAILURE);

}

fclose(shared\_file);

}

void \_send\_reply(int sock, short pid, struct message \* msg,

struct sockaddr\_in requester\_address) {

printf("\nSending timestamped reply to an outstanding file write request.\n");

/\* incrementing the timestamp of reply message\*/

msg->message\_ts += 1;

msg->message\_type = REPLY\_MSG;

printf("Message timestamp: %d\n", msg->message\_ts);

err\_handler(sendto(sock, msg, sizeof(struct message), 0,

(struct sockaddr\_in \*) &requester\_address,

sizeof(struct sockaddr\_in)), "SendReplyError");

}

struct request\_queue \* \_send\_release(int sock, struct message \* msg,

struct request\_queue \* front, struct sockaddr\_in addr) {

printf("\nExecution of critical section done.");

printf("\nSending release message to all other processes.\n");

/\* incrementing the message timestamp \*/

msg->message\_ts += 1;

msg->message\_type = RELEASE\_MSG;

printf("Message timestamp: %d\n", msg->message\_ts);

/\* remove the current process request from front of the queue \*/

if(front != NULL) {

struct request\_queue \* temp = front;

front = front->next\_request;

free(temp);

temp = NULL;

}

/\* broadcast the timestamped release message \*/

err\_handler(sendto(sock, msg, sizeof(struct message), 0,

(struct sockaddr\_in \*) &addr, sizeof(struct sockaddr\_in)), "SendReleaseError");

return front;

}

//This function receives the release message from the process writing to the shared file.

struct request\_queue \* \_recv\_release(int sock, short pid, struct request\_queue \*req) {

printf("\nWaiting for RELEASE message from the process writing to file.\n");

struct message recvd\_msg;

while(1) {

err\_handler(recvfrom(sock, &recvd\_msg, sizeof(struct message),

0, NULL, NULL), "RecvRlsError");

if(recvd\_msg.message\_type == RELEASE\_MSG && recvd\_msg.process\_id != pid) {

break;

}

}

short requester\_id = recvd\_msg.process\_id;

/\* looking for the request in the request queue of current process and deleting it\*/

struct request\_queue \* temp = req;

struct request\_queue \*temp2 = NULL;

if(temp != NULL) {

if(temp->requesting\_process\_id == requester\_id) {

req = temp->next\_request;

free(temp);

temp = NULL;

}

else {

while(temp->next\_request != NULL) {

if(temp->next\_request->requesting\_process\_id == requester\_id) {

temp2 = temp->next\_request;

temp->next\_request = temp->next\_request->next\_request;

free(temp2);

temp2 = NULL;

break;

}

temp = temp->next\_request;

}

}

}

printf("RELEASE Message Received.\n");

return req;

};

void \_exec\_process\_1() {

short process\_id = 1;

short permission[REQUEST\_SET+1] = {0}; /\* an array to store permissions from other processes\*/

/\* permission to execute CS from itself is always granted \*/

permission[process\_id - 1] = 1;

/\*

we declare a sockaddr\_in array of size equal to REQUEST\_SET, as these

many processes can request the current process for CS permission.

current process need to have their addresses, in order to reply to them.

\*/

struct sockaddr\_in requester\_address[REQUEST\_SET];

/\* address to which current process will bind to \*/

struct sockaddr\_in address, bcast\_address;

int sock = \_create\_sock\_and\_bind(&address, &bcast\_address);

/\*

socket is created and binded. Now process 1 broadcasts its request to write to the file.

we create the REQUEST message to be sent by process 1

\*/

struct message msg\_process\_1 = {0}; /\* initiate all values to zero \*/

/\* initiate the request queue and keep a pointer to the front of queue\*/

struct request\_queue \* front = NULL;

/\*Setting all addresses corresponding to requesting processes as zero\*/

for(int i = 0; i < REQUEST\_SET; i++) {

memset(&requester\_address[i], 0, sizeof(requester\_address[i]));

}

/\* send the request to execute critical section to all processes \*/

front = \_send\_cs\_request(sock, process\_id, &msg\_process\_1, front, bcast\_address);

/\*

after sending its own CS request, process 1 receives the CS request from

process 2 and 3. It receives them and stores in request\_queue

\*/

for(int i = 0; i < REQUEST\_SET; i++) {

front = \_recv\_cs\_request(sock, front, &requester\_address[i]);

}

/\* Till now, the process has sent its own CS request and received CS requests of all other

processes. Now, it will wait for reply from all other processes, in order to enter the CS.

The CS or critical section here is the shared file which every process wants to write. \*/

/\* getting the replies \*/

printf("\nWaiting for REPLY from all other processes.\n");

for(int i = 0; i < REQUEST\_SET; i++) {

\_recv\_reply(sock, process\_id, &msg\_process\_1, permission);

printf("Received REPLY %d\n", i+1);

}

/\*If the process's REQUEST timestamp is smaller than the received timestamp,

hopefully we will enter the critical section.\*/

printf("\nProcess %d REQUEST timestamp: %d\n", process\_id, msg\_process\_1.message\_ts);

/\*

after receiving the reply we check two conditions L1 and L2 of lamport's algorithm

and if the condition is satisfied, process 1 exceutes CS

\*/

short eligible\_for\_cs = checkLamportCondition(front, permission, process\_id);

if(eligible\_for\_cs == 1) {

startWritingToFile(process\_id);

}

/\* now the process will tell everyone that it's releasing the critical section \*/

front = \_send\_release(sock, &msg\_process\_1, front, bcast\_address);

/\* Now, process 1 has outstanding CS requests from all other processes

in the request set. It will now send a timestamped reply to these processes one by

one\*/

for(int i = 0; i < REQUEST\_SET; i++) {

\_send\_reply(sock, process\_id, &msg\_process\_1, bcast\_address);

/\* after sending the reply, we wait for the release message \*/

front = \_recv\_release(sock, process\_id, front);

}

printf("\nDONE.\n");

};

/\*

Code corresponding to process 2. Similar to \_exec\_process\_1(), however, the ordering of

sendto() and recvfrom() will be different.

\*/

void \_exec\_process\_2() {

short process\_id = 2;

int bytes\_written;

short permission[REQUEST\_SET+1] = {0}; /\* an array to store permissions from other processes\*/

/\* permission to execute CS from itself is always granted \*/

permission[process\_id - 1] = 1;

/\*

we declare a sockaddr\_in array of size equal to REQUEST\_SET, as these

many processes can request the current process for CS permission.

current process need to have their addresses, in order to reply to them.

\*/

struct sockaddr\_in requester\_address[REQUEST\_SET];

/\* address to which current process will bind to \*/

struct sockaddr\_in address, bcast\_address;

int sock = \_create\_sock\_and\_bind(&address, &bcast\_address);

/\*

socket is created and binded. Now process 1 broadcasts its request to write to the file.

we create the REQUEST message to be sent by process 1

\*/

struct message msg\_process\_2 = {0}; /\* initiate all values to zero \*/

/\* initiate the request queue and keep a pointer to the front of queue\*/

struct request\_queue \* front = NULL;

/\*Setting all addresses corresponding to requesting processes as zero\*/

for(int i = 0; i < REQUEST\_SET; i++) {

memset(&requester\_address[i], 0, sizeof(requester\_address[i]));

}

front = \_recv\_cs\_request(sock, front, &requester\_address[0]);

/\* send the request to execute critical section to all processes \*/

front = \_send\_cs\_request(sock, process\_id, &msg\_process\_2, front, bcast\_address);

front = \_recv\_cs\_request(sock, front, &requester\_address[1]);

\_send\_reply(sock, process\_id, &msg\_process\_2, requester\_address[0]);

front = \_recv\_release(sock, process\_id, front);

printf("\nWaiting for REPLY from all other processes.\n");

for(int i = 0; i < REQUEST\_SET; i++) {

\_recv\_reply(sock, process\_id, &msg\_process\_2, permission);

printf("Received REPLY %d\n", i+1);

}

/\*If the process's REQUEST timestamp is smaller than the received timestamp,

hopefully we will enter the critical section.\*/

printf("\nProcess %d REQUEST timestamp: %d\n", process\_id, msg\_process\_2.message\_ts);

short eligible\_for\_cs = checkLamportCondition(front, permission, process\_id);

if(eligible\_for\_cs == 1) {

startWritingToFile(process\_id);

}

/\* now the process will tell everyone that it's releasing the critical section \*/

front = \_send\_release(sock, &msg\_process\_2, front, bcast\_address);

\_send\_reply(sock, process\_id, &msg\_process\_2, bcast\_address);

front = \_recv\_release(sock, process\_id, front);

printf("\nDONE.\n");

};

void \_exec\_process\_3() {

short process\_id = 3;

int bytes\_written;

short permission[REQUEST\_SET+1] = {0}; /\* an array to store permissions from other processes\*/

/\* permission to execute CS from itself, is always granted \*/

permission[process\_id - 1] = 1;

struct sockaddr\_in address, bcast\_address;

int sock = \_create\_sock\_and\_bind(&address, &bcast\_address);

for(int i = 0; i < REQUEST\_SET; i++) {

memset(&requester\_address[i], 0, sizeof(requester\_address[i]));

}

for(int i = 0; i < REQUEST\_SET; i++) {

front = \_recv\_cs\_request(sock, front, &requester\_address[i]);

}

front = \_send\_cs\_request(sock, process\_id, &msg\_process\_3, front, bcast\_address);

for(int i = 0; i < REQUEST\_SET; i++) {

\_send\_reply(sock, process\_id, &msg\_process\_3, bcast\_address);

/\* after sending the reply, we wait for the release message \*/

front = \_recv\_release(sock, process\_id, front);

}

printf("\nWaiting for REPLY from all other processes.\n");

for(int i = 0; i < REQUEST\_SET; i++) {

\_recv\_reply(sock, process\_id, &msg\_process\_3, permission);

printf("Received REPLY %d\n", i+1);

}

printf("\nProcess %d REQUEST timestamp: %d\n", process\_id, msg\_process\_3.message\_ts);

short eligible\_for\_cs = checkLamportCondition(front, permission, process\_id);

if(eligible\_for\_cs == 1) {

startWritingToFile(process\_id);

}

/\* now the process will tell everyone that it's releasing the critical section \*/

front = \_send\_release(sock, &msg\_process\_3, front, bcast\_address);

printf("\nDONE.\n");

};

//Main function.

int main(int argc, char \* argv[]) {

short process\_id;

if(argc != 2) {

if(argc < 2) {

printf("\nToo few arguments.\n");

}

else {

printf("\nToo many arguments.\n");

}

printf("\nUsage: ./lamport-dme -p<1|2|3>\n\n");

exit(EXIT\_FAILURE);

}

/\*ensure process\_id argument is in proper format\*/

if(strlen(argv[1]) != 3) {

printf("\nInvalid Argument.\n");

printf("\nUsage: ./lamport-dme -p<1|2|3>\n\n");

exit(EXIT\_FAILURE);

}

process\_id = atoi(argv[1] + 2); /\*get 3rd character from the argument as process\_id\*/

if(process\_id == 0 || process\_id > 3) {

printf("\nInvalid Argument.\n");

exit(EXIT\_FAILURE);

}

if(process\_id == 1) {

\_exec\_process\_1();

}

else if(process\_id == 2) {

\_exec\_process\_2();

}

else {

\_exec\_process\_3();

}

return 0;

}

## 

## 

## 

## 

## 