Raymond’s MPI algorithm

Author: Dizhen Liang (31240291)

Introduction

Distributed Mutual Exclusion

- Mutual exclusion, also known as mutex, is a concept in concurrent programming.
- It is a program object that prevents simultaneous access to a shared resource.
- This concept is crucial when multiple processes need to access a shared resource (critical region), like a data object.

Problem Statement

- The goal is to avoid situations where multiple entities are trying to modify the data simultaneously, especially for the large distributed system.
- Race problem would be incurred as the result.
- Proposing an efficient and stable way of managing the large number of processes to modify the shared data would be vital for the large distributed system to work effectively.

Related Works

There are two basic approaches for distributed mutual exclusion:

- Token based approach
- Non-token approach

Non-token approach:

All the approach other than circulating token to access critical region

- Central Server Algorithm

Token-based approach:

The process with the token have the privilege to access critical region

- Ring Algorithm
- Raymond's Algorithm

Central Sever Algorithm

- One process in the distributed system is elected as the coordinator.
- All shared data is maintained by this central server.
- When a process wants to modify data in a critical region -> sends a request message to the central sever node.
- The central sever would receive the data via the send then update the corresponding data and return acknowledgement messages.

Advantage:

- Algorithm is simple to implement

Limitation:

- Central server become a bottleneck for the large distributed systems.

Improvements:

- shared data can be distributed among several servers.

Ring Algorithm

Processes are organized in a logical ring and communicate with one neighbour. The token travels along the ring, granting access to the shared resource.

Advantages:

- No need for central sever
- Ideal for a large distributed system

Limitations:

- Uni-directional ring -> token must be passed through all nodes
- On node down -> whole network down

Methodology - Raymond's Algorithm

To solve the problem proposed in problem statement, Raymond's Algorithm might be the suitable solution.

Raymond's Algorithm:

- Token-passing algorithm for tree-based networks.
- Processes are connected through a tree topology.
- Each process only communicate to its parent node
- Token is passed between them through MPI Communicator
- The node holding the token serves as the root of the tree and execute operation to the critical region

Analysis - Raymond’s Algorithm

Advantages:

- No central server required -> no bottleneck by a single sever
- More flexible -> no need to pass the token around the ring
- Guaranteed to be O (log n) complexity per critical region access if the processes are allocated into a K-ary tree

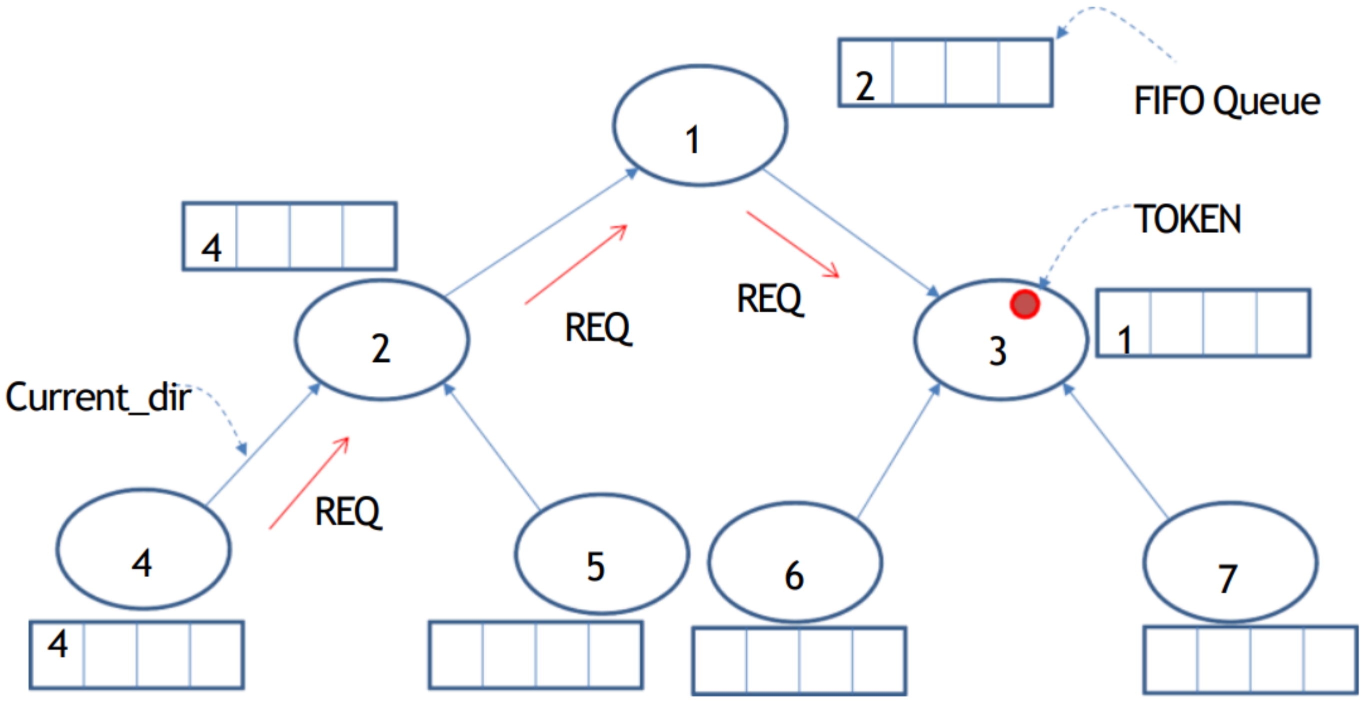
Limitations:

- Requires 2 * (Longest path length of the tree) message invocation per critical section entry.

Conclusion:

Raymond’s Algorithm has higher flexibility, efficiency and stability compared with the Central Server Algorithm and Ring Algorithm as long as all nodes or processes are arranged in K-ary tree topology.

Representation – Raymond's Algorithm



Actively send request

```c
void *port_func(void *arg)
{
    // Cast the argument to a pointer to a port_t struct
    port_t *port = (port_t *)arg;

    // Use the port pointer
    srand(port->id + port->node->id);

    // Set availability to 0
    port->available = 0;

    // Update on the port array with the availability
    for(int i= 0; i < sizeof(port->node->ports)/sizeof(port->node->ports[0]); i++)
    {
        // // //Randomly set availability to 0 or 1, but mostly 0
        port->node->ports[i] = (rand() % 10 < 9) ? 0 : 1;

    }
    // Sleep for 10 seconds
    sleep(10);

    return NULL;
}
```

```c
int occupied = 0;
for (int i =0; i < K; i++)
{
    if (node.ports[i] == 0)
    {
        occupied++;
    }
}
//Actively send request if condition fulfilled
//count for itself or it's neighbours that whether they are vacant
int vacancies = 0;
double rate = (double) occupied / K;
// printf("\n %d has rate: %f \n", node.id, rate);
// fflush(stdout);
if (rate >=  THRESHOLD)
{
    //making request through the parents until the holder(toppest parent node)

    //push to make it no empty by pushing itself into the queue
    if(!node.asked){
        node.req_q.push(node.id);
        printf("\n %d push %d into queue\n", node.id, node.req_q.front());
        fflush(stdout);
    }

    //make request to the holder
    //put request node itself into the queue
    //each process send request until it reach the holder node
    //stop it from repetively sending request, if already send out one
    if(node.holder != node.id && !node.req_q.empty() && !node.asked){

        //send request if condition in single port fulfilled
        //do following Recv for the Send if need to get immediate reponse
        if(node.parent != PARENT)
        {
            MPI_Send(&node.id, 1, MPI_INT, node.parent, REQTAG, comm2D); // Access members via pointer
            printf("\n %d send request to: %d \n", node.id, node.parent);
            fflush(stdout);
            node.asked = true; //sent request = asked
        }
```

Receive request then forward request

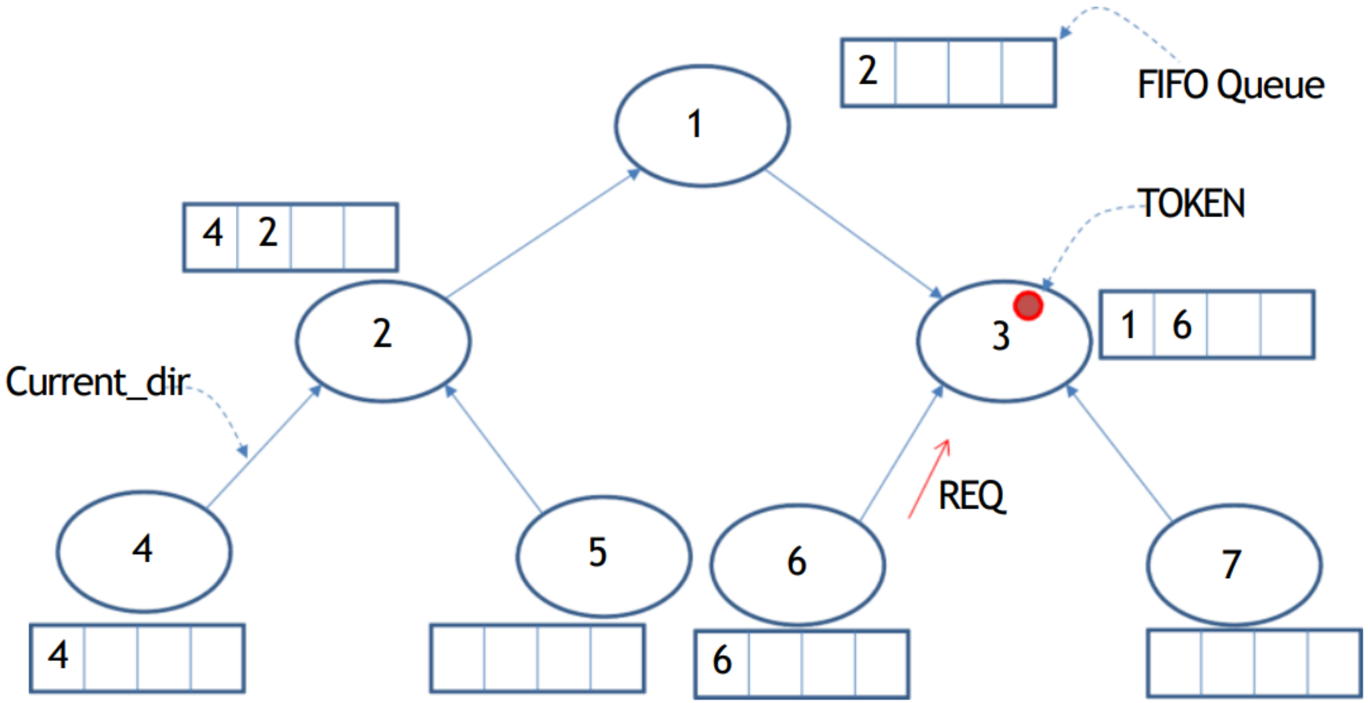
```
}
//After receive request
MPI_Iprobe(MPI_ANY_SOURCE, REQTAG, comm2D, &flag, &status);
if(flag)
{
    int source;                         //source from the status

    MPI_Recv(&source, 1, MPI_INT, status.MPI_SOURCE, REQTAG, comm2D, &status);
    printf("\n Recv %d push %d into queue\n", node->id, source);
    fflush(stdout);
    node->req_q.push(source);

    //If the source if not the holder, continue to send to its parent
    //stop sending if node.id = node.holder

    if(node->id != node->holder && !node->req_q.empty() && !node->asked){
        node->asked = true;
        if(node->parent != PARENT){
            MPI_Send(&node->id, 1, MPI_INT, node->parent, REQTAG, comm2D);
            printf("Recv-Send, id: %d to parent: %d", node->id, node->parent);
            fflush(stdout);
        }
```

Before Sending Token



Send token and forward request

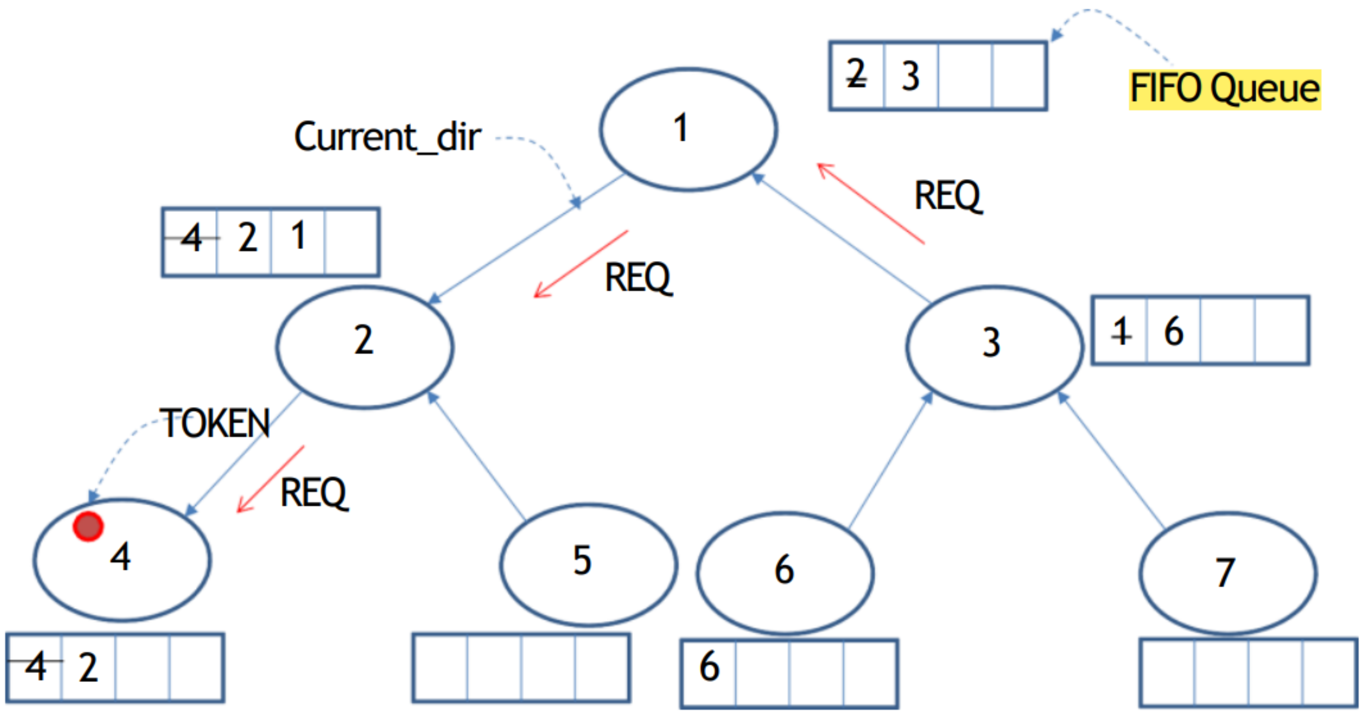
```
int ori_p = node.parent;
node.parent = node.holder;
printf("\n Rank: %d, original parent: %d -> new parent: %d", node.id, ori_p, node.parent);
fflush(stdout);

//Send token alone to let them execute for FIFO
MPI_Send(&node.id, 1, MPI_INT, node.parent, TOKENTAG, comm2D);

//since sending out token
node.req_q.pop();
node.asked = 0;
printf("\nid: %d pop out %d", node.id, node.holder);
fflush(stdout);

if (!node.req_q.empty()){
    MPI_Send(&node.id, 1, MPI_INT, node.parent, REQTAG, comm2D); // Access members via pointer
    printf("\n %d forward request to: %d \n", node.id, node.parent);
    fflush(stdout);
}
```

After Sending Token



Request node receive token

```
//Token handler - After receive token
MPI_Iprobe(MPI_ANY_SOURCE, TOKENTAG, comm2D, &flag, &status);
if(flag)
{
    int token_source;                    //source from the status
    //store the request process to the source
    //receive the token from another process
    MPI_Recv(&token_source, 1, MPI_INT, status.MPI_SOURCE, TOKENTAG, comm2D, &status);
    printf("\n rank: %d receive token from %d \n", node->id, token_source);
    fflush(stdout);
    node->holder = node->id;

}
```

```
while(1)
{
    //handle the use of token                          //has pending request
    //At the begining, random node holding the token
    //In my case, node 0 holds it
    //if some node other than it make request
    //reponse request by passing down the token since itself is the toppest parent
    if (node.holder == node.id && !node.isUsing && !node.req_q.empty()){
        node.holder = node.req_q.front();

        if(node.holder==node.id){
            node.isUsing = true;

            shared_data[node.id]++;
            // MPI_Win_unlock(0, win);
            printf("\n rank: %d, shared data: %d", node.id, shared_data[node.id]);
            time_t t = time(NULL);
            char* time_str = ctime(&t);
            //make it a-append so it can append not overwrite and create a logFile if is not
            FILE *logFile = fopen("log.txt", "a");  //s come with newline

            //Log into file
            fprintf(logFile,"\nLogged time: %s",time_str);
            fprintf(logFile, "Process: %d with Count: %d\n",node.id, shared_data[node.id]);
            fclose(logFile);

            //after usage, back to original state
            node.req_q.pop();//pop out itself, since already done the modification
            node.asked = 0;
            printf("\n id: %d pop out %d", node.id, node.holder);
            fflush(stdout);
            node.isUsing = 0;
            node.ports[0] = 1;
        }
```

Results

```
Logged time: Sat Nov  4 15:35:07 2023
Process: 5 with Count: 1

Logged time: Sat Nov  4 15:35:10 2023
Process: 1 with Count: 1

Logged time: Sat Nov  4 15:35:12 2023
Process: 4 with Count: 1

Logged time: Sat Nov  4 15:35:13 2023
Process: 3 with Count: 1

Logged time: Sat Nov  4 15:35:16 2023
Process: 0 with Count: 1

Logged time: Sat Nov  4 15:35:18 2023
Process: 2 with Count: 1

Logged time: Sat Nov  4 15:35:20 2023
Process: 2 with Count: 2

Logged time: Sat Nov  4 15:35:22 2023
Process: 0 with Count: 2

Logged time: Sat Nov  4 15:35:24 2023
Process: 2 with Count: 3
```

Reference

- De Turck, F. (2022). Comparison of algorithms for distributed mutual exclusion through simulations. Retrieved from https://arxiv.org/abs/2211.01747v2

- Raymond, K. (1989). A tree-based algorithm for distributed mutual exclusion. ACM Transactions on Computer Systems, 7(1), 61-77.