HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
BBM342 OPERATING SYSTEMS
PROJECT I

Eylül Tuncel
21727801

1. INTRODUCTION

In this project we expected to create interprocess communication and synchronization mechanisms. In this project, the minimal and simplified set for MPI is implemented for shared memory multiprocessor architectures.

2. THE PROJECT

In the project, MPI_Send and MPI_Recv functions are used for exchanging data. Both send and receive (i.e. Rendezvous) are blocking functions. The communication is successful if the source rank is matched with the destination rank and both tag values are the same. We are given some function declarations:

```
int MPI_Init(int *argc, char ***argv);
int MPI_Finalize();
int MPI_Comm_size(int *size);
int MPI_Comm_rank(int *rank);
int MPI_Recv(void *buf, int count, int datatype, int source, int tag);
int MPI_Send(const void *buf, int count, int datatype, int dest, int tag);
```

For synchronization, I try to implement the producer consumer problem.There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them. For implementation of rendezvous type of synchronization, both sender and receiver processes become producer and consumer repeatedly. First sender process waits for a receiver and then when the receiver process starts, it waits for the sender. In this way they both wait for each other and become rendezvous.

For shared memory implementation, ı create separate small shared memories for each process. Each process reads from its own shared memory and when it wants to send a message to another process, writes to their special shared memory locations.

3. IMPLEMENTATION

3.1. MPI_Init

In this function I initialize empty, full and mutex semaphores special to the processes. And I initialize, open shared memory for each of the processes. Also assign values to the “rank” and “size” global variables.

Global variables :

- RANK = rank of the process
- SIZE = number of the total processes

For each process the names and initialized values of semaphores are :

- semaphore_empty{rank} = n (buffer size=10)
- semaphore_full{rank} = 0
- mutex = 1

Name of the shared memory special to the process:

- shared_memory{rank}

3.2. MPI_Recv

In the receive function, first I open the source process's semaphores as "empty_source" and "full_source". It gives me the ability to reach to the source process semaphores and use them. (it is necessary for producer consumer type problem)

Then the receive function signals to the sender process (because the sender process waits for a receiver to start receiving function). After signaling, the receiver starts waiting for the sender process to write its message to the shared memory. When the sender finishes his job with shared memory it signals "full" semaphore and the receiver process can start reading messages from its shared memory. If tag values are equal then the receiver process signals to the sender and finishes its execution. If tag values are not equal it results with deadlock. Receiver never signals to the sender process and the sender can never finish its execution. Waits for a signal infinitely.

```
// first signal the sender process for start their execution
sem_wait(empty_source);
sem_post(full_source);

// wait for sender process to send message
sem_wait(full);
//for critical operations mutex must be waited
sem_wait(&mutex);

// read message from shared memory
memcpy(buf, pointer, strlen(pointer)+1);
pointer += count*datatype;

// if tag value (in the shared memory) is equal to the tag of the receiver
if(tag == atoi(pointer)){

    // then send signal to the sender process for let it continue and finish
    sem_post(&mutex);
    sem_post(empty);
    sem_post(full_source);

}
```

3.3. MPI_Send

In the sender function, first I open the destination process's semaphores as "empty_dest" and "full_dest". It gives me the ability to reach to the destination process semaphores and use them. (it is necessary for producer consumer type problem). After that I open the destination process's shared memory location, because the sender wants to write a message to the sender process. (ın my implementation every process receives messages from its own shared memory locations.)

Then the sender process first waits for a receiver process to start. When the receiver process starts execution it first signals full semaphore.(it means there is a receiver started waiting for the sender now). After that the sender process can start writing messages to the receiver process's shared memory. When it's done with writing, the sender process signals to the receiver process (receiver process understands that it can start reading). When it's all done, the sender process starts waiting for the receiver process to finish its reading and signals to sender. The last line means, sender process can never finish before the receiver successfully reads messages. If there is a mismatch in tags then two processes can enter a deadlock.

```
// first wait for the receiver process
sem_wait(full);
sem_post(empty);

// when receiver process starts execution, sender can continue to produce message
sem_wait(empty_dest);
//for critical operations mutex must be waited
sem_wait(&mutex);

// write message to the destination shared memory
memcpy(pointer, buf, strlen(buf)+1);
pointer += count*datatype;
// write tag value to the destination shared memory
sprintf(pointer,"%d",tag);
pointer += sizeof(int);

// then send signal to the receiver process for let it continue and finish
sem_post(&mutex);
sem_post(full_dest);

// wait for receiver to finish its execution, and then sender can be finished
sem_wait(full);
```

3.4. MPI_Comm_size

This function returns the total number of processes running in our system.

3.5. MPI_Comm_rank

This function returns the rank of the process, ranging from 0 to n (n=total processes) .

3.6. MPI_Finalize

In this function, I destroy all the semaphores created and unlink the shared memory of the process.

4. TEST CASES

4.1. **Test1** (from Test.pdf)

```
int main(int argc, char *argv[], char *environ[])
{
int npes,myrank,number,i;
MPI_Init(&argc, &argv);
MPI_Comm_size(&npes);
MPI_Comm_rank(&myrank);
if(myrank == 0){
         for(i = 1; i < npes ; i++){
             MPI_Recv(&number, 1, sizeof(int), i, 0);
             printf("From process %d data= %d, RECEIVED!\n",i,number); }
     } else  {
         number=myrank;
         MPI_Send(&number, 1, sizeof(int), 0, 0); }
     if(myrank == 0){
         for(i = 1; i < npes ; i++) {
             MPI_Send(&i, 1, sizeof(int), i, 0); }
     } else {
         MPI_Recv(&number, 1, sizeof(int), 0, 0);
         printf("RECEIVED from %d data= %d, pid=%d \n",myrank,number,getpid());}
    for(i = 0; i < 10000 ; i++) {
        if(myrank%2 == 0) {
            MPI_Recv(&number, 1,sizeof(int), (myrank+1)%npes, 0);
            MPI_Send(&number, 1,sizeof(int), (myrank+1)%npes, 0);
        } else {
            MPI_Send(&number, 1,sizeof(int), (myrank-1+npes)%npes, 0);
            MPI_Recv(&number, 1,sizeof(int), (myrank-1+npes)%npes, 0);}
        }
    printf("FINISHED%d\n",myrank);
    MPI_Finalize();  }
```

- ➔ In this test, first all the processes other than process0 are sending messages to the process0 .All processes send their ranks as a message. Process0 is receiving messages from all other processes messages.
- ➔ Then process0 is sending messages to all other processes.
- ➔ And at the end there is a stress test that works for 10000 times. All processes send and receive messages in pairs.

My output on dev:

```
-bash-4.2$ gcc main.c -o main -lpthread -lrt
-bash-4.2$ gcc my_mpirun.c -o my_mpirun -lpthread
-bash-4.2$ ./my_mpirun 10 main
From process 1 data= 1, RECEIVED!
From process 2 data= 2, RECEIVED!
From process 3 data= 2, RECEIVED!
From process 4 data= 2, RECEIVED!
From process 5 data= 3, RECEIVED!
From process 6 data= 5, RECEIVED!
From process 7 data= 6, RECEIVED!
From process 8 data= 6, RECEIVED!
From process 9 data= 7, RECEIVED!
RECEIVED from 1 data= 1, pid=27273
RECEIVED from 2 data= 2, pid=27274
RECEIVED from 3 data= 3, pid=27275
RECEIVED from 4 data= 4, pid=27276
RECEIVED from 6 data= 6, pid=27278
RECEIVED from 5 data= 5, pid=27277
RECEIVED from 8 data= 8, pid=27280
RECEIVED from 7 data= 7, pid=27279
RECEIVED from 9 data= 9, pid=27281
FINISHED7
FINISHED6
FINISHED1
FINISHED0
FINISHED3
FINISHED2
FINISHED9
FINISHED8
FINISHED5
FINISHED4
```

First all processes send data to the process0. After process0 receives it it prints out messages with the ranks of the sender processes.

After that all processes receive messages from process0. And receiver processes print out a message.

And lastly all processes send messages in pairs 10000 times. When this send-receive loop finishes successfully, the processes exit. And print out “FINISHED{rank}” message.

4.2. **Test2** (from Test.pdf)

```
int main(int argc, char *argv[], char *environ[])
{
int npes,myrank,number,i;
MPI_Init(&argc, &argv);
MPI_Comm_size(&npes);
MPI_Comm_rank(&myrank);
if(myrank == 0) {
        MPI_Recv(&number, 1, sizeof(int), 1, 0);
        printf("RECEIVED!\n");
    } else {
        MPI_Send(&myrank, 1, sizeof(int), 0, 0);
        printf("SENT!\n"); }
    printf("STAGE 2!\n");
    if(myrank == 0)  {
        MPI_Recv(&number, 1, sizeof(int), 1, 1);
```

```
        printf("RECEIVED >>> 2\n");
    } else {
        MPI_Send(&myrank, 1, sizeof(int), 0, 0);
        printf("SENT >>> 2\n");  }
    printf("FINISHED%d\n",myrank);
    MPI_Finalize();
}
```

In this test case, a pair of processes send and receive messages in between. But there is a message with a mismatched tag. When the process receives mismatched tag, the system enters deadlock.

```
-bash-4.2$ gcc my_mpirun.c -o my_mpirun -lpthread
-bash-4.2$ gcc main.c -o main -lpthread -lrt
-bash-4.2$ ./my_mpirun 2 main
RECEIVED!
STAGE 2!
SENT!
STAGE 2!
RECEIVED >>> 2
FINISHED0
```

As you can see in my output, the process enters deadlock state. Therefore it can never be finished. One message is send and received but the second one leads to deadlock because of mismatched tag.