National University
of computer and emerging sciences

FAST Islamabad Campus

Assignment #1 Report:

Interprocess Communication

CS2006 – Operating Systems

Group Members:

Sarim Aeyzaz (21I-0328)
M.Saifullah Amin (21I-2717)
Hafiz Hammad Ahmed (21I-0343)
Ehtsham Walidad (21I-0260)

Part 1:

Problem Statement:

The problem explicitly requires the proper management of activity between two processes, denoted as "P1" and "P2." One process is responsible for collecting four roll numbers, while the other process is in charge of extracting the last four digits, respectively.

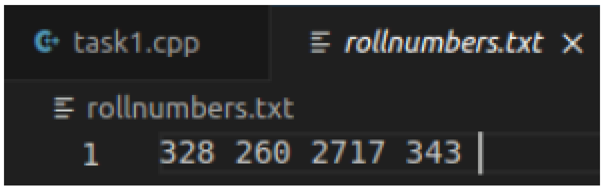
Solution:

In order to tackle this problem we will use a combination of pipes and fork. Explicitly we would be using a “childPipe” and a “parentPipe” these will be used for data transfer between the two processes.

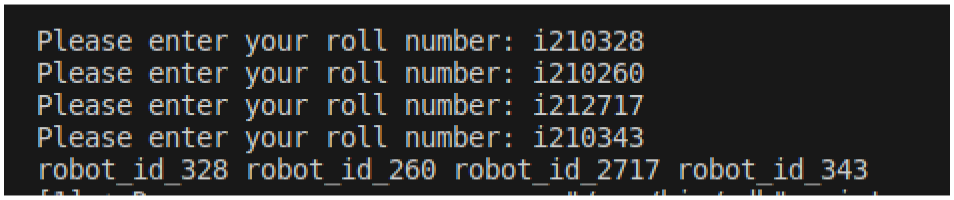
Firstly , in the child process we prompt the user to enter the roll numbers , which are stored in a roll number array. This array is then sent to the parent process via the childPipe.

The parent process receives the roll numbers from the child process and then extracts the last 4 digits and stores in the “numbersOnly” array. This is then subsequently sent back to the child process via the parentPipe.

Finally the parent process writes the received digits into a file and closes it. Please note that apt error checking is implemented at each step.



```
328 260 2717 343
```

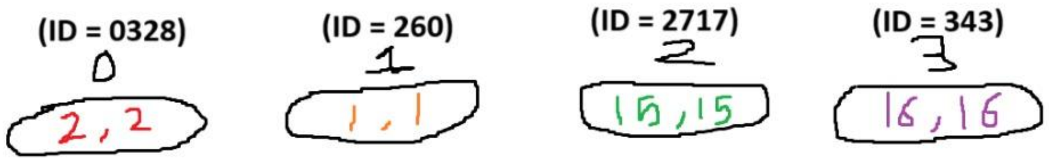


```
Please enter your roll number: i210328
Please enter your roll number: i210260
Please enter your roll number: i212717
Please enter your roll number: i210343
robot_id_328 robot_id_260 robot_id_2717 robot_id_343
```

Part 2:

Problem statement:

The problem basically requires proximity based communication between 4 robots in an imaginary plane. Each robot in this plane has specific x and y coordinates, each robot has a certain distance from each other based on their relative coordinates. But to assert this, all 4 of the robots must be communicating with each other actively. The identifiers (0, 1, 2, …, n-1) of n total robots will have a read/write sequence as follows:



Read Write Order (Based on Identifier)

```
W - R - R - R
R - W - R - R
R - R - W - R
R - R - R - W
```

Solution: (with Named Pipes)

We define that there are 4 robots , but along with this we also instantiate 4 files for each robot. Each file contains a unique "identifier" which defines the robot's position in the read/write sequence explained further below. Each robot has the facility to store its own coordinates and in addition to this the coordinates of all other robots in the plane.

First of all the robot generates all the possible labels for the pipes that are accessible to that specific robot. For instance robot 2 would have the facility to write to robot 0, 1, and 3. Similarly , robots 0, 1 and 3 would be able to read from robot 2. The labels for these communication pipes are displayed below:

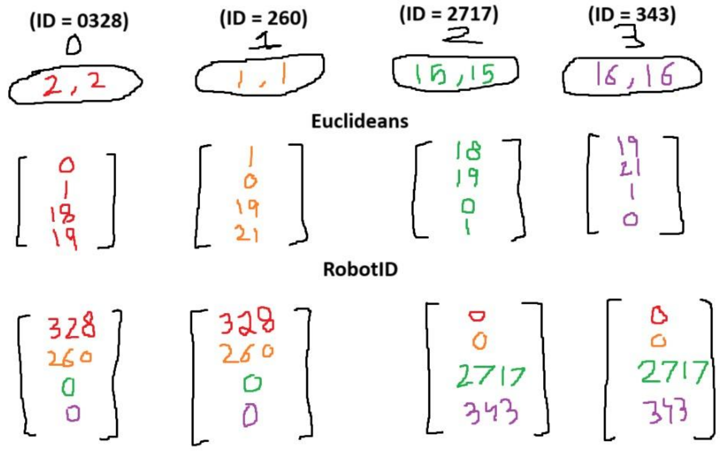
```
// [2-to-0, 2-to-1, 2-to-2, 2-to-3] for writeLabels
// [0-to-2, 1-to-2, 2-to-2, 3-to-2] for readLabels
```

After all the pipe definitions are complete the robot moves onto initializing its own ID from the “rollnumbers.txt” that we generated in Task1. Please note adept error checking is implemented at this step.

Following this the robot prompts the user to enter its new coordinates. Here , entry validation is performed in order to make sure that the coordinates that are being entered are inclusive between 1 and 30 and are clearly integers.

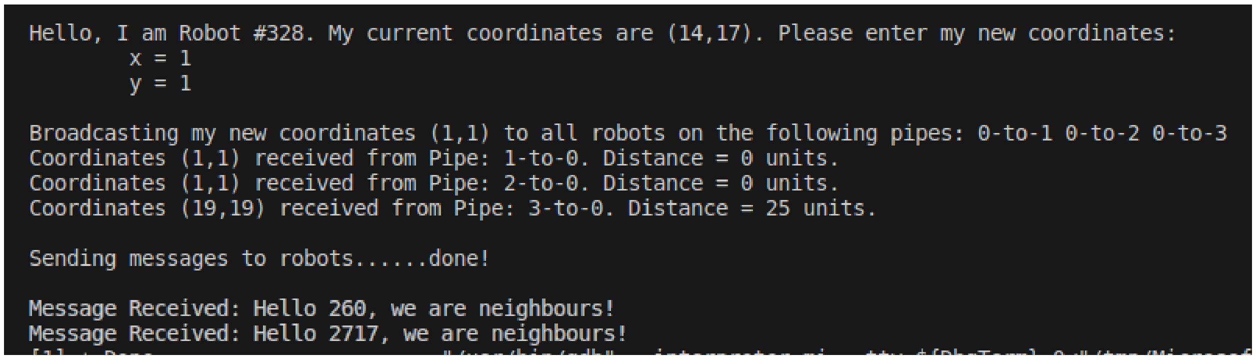
Once the above steps are successfully completed, we move onto the actual communication of the robots themselves. We must understand that for each of the 4 robots they both have two modes: “Writing” and “Reading”. When a bot is in writing mode it broadcasts its coordinates to every other robot using the “writeLabels” array in order to iterate through all the possible pipes that the specific robot has to take. In the condition that robot is not in the writing mode, it is instead reading the coordinates from the other robots in the plane. Once received, the Euclidean distance is calculated between the two bots and stored within an array which defines the distance of the current robot with all other robots. Each robot calculates its own Euclidean distance list. This would notify the robot to whom it should broadcast its ID to.

So by this point the above steps are complete for all 4 robots:



When outputting, ignore your own RobotID and 0's in the list

As shown above by this point all 4 robots have a list of the distances between them and the other robots. At this point the robot checks which other robot has a distance of less than 10. If they do, then the current robot sends its own RobotID to its neighbors , else it just sends a “0” signifying that the specific robot is not a neighbor.



```
Hello, I am Robot #328. My current coordinates are (14,17). Please enter my new coordinates:
        x = 1
        y = 1

Broadcasting my new coordinates (1,1) to all robots on the following pipes: 0-to-1 0-to-2 0-to-3
Coordinates (1,1) received from Pipe: 1-to-0. Distance = 0 units.
Coordinates (1,1) received from Pipe: 2-to-0. Distance = 0 units.
Coordinates (19,19) received from Pipe: 3-to-0. Distance = 25 units.

Sending messages to robots......done!

Message Received: Hello 260, we are neighbours!
Message Received: Hello 2717, we are neighbours!
```

Once it's been identified which bots are the neighbors of the current bot a message is received from them symbolizing that they are each other's neighbors.

Solution: (with Shared Memory):

The difference between both the solutions is simply in the way that the communication is managed between the robots. Where they are no longer entirely dependent on the pipes for communication but instead use a shared memory system.

To do this we create keys out of the previously mentioned “writeLabels” , and using these keys we create a shared memory segment in our OS which can only be accessed by that specific key. This means that 0-to-1 , 0-to-2 etc which were previously just pipes are now shared memory segments.

Now in the writing mode instead of broadcasting the coordinates via the pipes instead we send a dummy variable into the pipe and in that same code section we write into the shared memory segment. So in actuality the data transfer is done via the shared memory but the pipes are there for the purpose of synchronization and act as a pseudo-interrupt. This ensures that the order of how the robots communicate with each other is maintained and no garbage values are read in the shared memory segments.

Similarly in the reading mode the pipe only really reads the dummy variable that we sent. We use the “readLabels” array to initialize and access the shared memory and feed them into the “RobotCoords” array. The rest of the procedure is identical to what’s outlined above.

National University
of computer and emerging sciences

Modified Problem (Asked by Sir)

The modified problem states that each robot is freely able to ask for coordinates however many times it wishes and the other robots in the environment keep listening to the changing coordinates and aptly display messages whether or not the former robot is now considered a neighbor.

Our first approach to solving this problem was just re-using our solution and letting it work for N number of inputs. However, it was incredibly difficult to figure out the read/write sequences between the robots when the broadcasting is set to be dynamic. As a result, we had to abandon this approach

Our next approach was relying on shared memory segments and noticing changes within the shared memory to update robots on whether a new neighbor has been found or not. However, no matter how many different variations we tried, each robot wouldn’t instantly receive the message and had to go through its own I/O operation before any message passing could be done. This resulted in an insufficient solution as well.

Our third approach was forking each robot, let the child process do the writing, and the parent process do the reading. This unfortunately still didn’t work because the terminal started to bug out. Sometimes it would be stuck in waiting for input before message passing and other times it wouldn’t even allow you to enter anything into the terminal. We reached a dead-end with this method and discarded it. Also, managing variable changes between the parent and child sections was a headache.

Our final attempt was using the pselect library to try to read input for a fixed number of time, otherwise, be stuck in busy waiting for coordinate changes in the shared memory. This was some-what of a good fix. However, there were massive synchronization issues between the robots. We tried to synchronize them with sleep functions however, it just wasn’t enough.

Therefore, we just decided to stick with our current submitted solution. If using threads were allowed then hopefully we could’ve tackled this problem with an appropriate solution which used threads.