1 Shared memory vs. message passing

1.1 Comparison

Shared memory allows multiple processes to read and write data from the same location. Message passing is another way for processes to communicate: each process can send messages to other processes.

Make a comparison between shared memory and message passing: where are they different and where are the similar? You might consider different models of message passing, for example with or without message loss.

Solution

Some ideas are:

- **Delay**: There is no delay with shared memory, if one process writes the other processes can read immediately. With message passing delay can happen (not necessarily in every model), different messages may even have different delays.

- **Overriding**: With shared memory if a process writes to a register, another process may override the value before anyone could read the register. In message passing this cannot happen. On the other hand messages may be lost, or the inbox buffer of a process may overflow, leading to similar results.

- **Consistency**: With message passing several messages may be sent at the same time, and the order of arriving message may be messed up. With shared memory the value of a register is always the value that was written last.

1.2 Examples

Consider the actions described below, in which model (shared memory or message passing) can you best describe them? Why and how?

- Communication via postcard
• Speaking in a room with two people
• Instant messages via Skype (data remains on client if partner is offline)
• Speaking in a room full of people

Solution

Postcard Clearly message passing. Messages may be lost, the order may be inconsistent and the inbox may overflow.

Two people speaking Shared memory: one can speak (=write), the other can listen (=read). Both speaking does not work, both listening neither.

Skype This could be modeled with message passing or with shared memory.

  Message passing: the text to send is a message, it cannot be lost and their order is consistent. However, it would be stored on the senders site until the reciever is online.

  Shared memory: there could also be an “inbox” register for each participant. The sender writes into the others inbox register. The reciever clears its register once it has seen the new text, the sender could then write into the register again.

Many people speaking Message passing: everyone is connected with everyone. If speaking message are sent not only to the intended listener(s) but also randomly to other people. If someone speaks loud, more messages are sent. The inbox of any person has limited size, once it is full arriving messages begin to override older messages. This models the fact that one cannot listen to many speakers at the same time.

2 Writing to multiple registers at the same time

A n-register allows up to n registers to be written at the same time. Processes may still only read one value at a time. Let n = 6, give a protocol which solves consensus for 3 processes. You may assume the registers are initialized with -1 and processes have a unique id.

Hints: You don’t need more than 6 registers (or one 6-register). You don’t need to write into all registers, you can write into a subset (e.g. you can atomically write into 3 registers). Compare pairs of processes, find out which process is the fastest.

Solution

We require 6 registers. We call the first three registers R₀, R₁ and R₂. To the other three registers we give the names R₀₁, R₀₂ and R₁₂. The goal is to find the fastest process and take it’s input value as decision.

In words, the protocol works as follows:

1. In a single step process 𝑖 writes its id into R𝑖 and into R𝑖,𝑗 for 𝑖 ≠ 𝑗.
2. It then checks for all 𝑖 > 𝑗 whether process 𝑖 was faster than process 𝑗:
   - If R𝑖,𝑗 = −1 then neither 𝑖 nor 𝑗 have yet done anything.
   - Otherwise, if R𝑖 = −1 then process 𝑗 must be faster than 𝑖.
   - Otherwise, if R𝑗 = −1 then process 𝑖 must be faster than 𝑗.
Otherwise $R_{i,j}$ holds the id of the process which was slower.

3. With all this information, a process can calculate which process must be the fastest one.

Solution in pseudocode

Written in pseudocode the protocol looks like this:

```plaintext
initialize()
{
    // R are the shared registers
    R[] = [-1, -1, -1, -1, -1, -1];

    // the input, an array of length 3
    input[] = [random(), random(), random()];
}

decide()
{
    id = this.getThreadId();

    // the identifiers of the other processes
    others = [ {0,1,2} without {id} ];

    // atomically write three registers
    write( R[id] = id, R[id,others[0]] = id, R[id,others[1]] = id );

    // pairwise comparison of process–speed
    fastest01 = faster( 0, 1, id );
    fastest02 = faster( 0, 2, id );
    fastest12 = faster( 1, 2, id );

    // find the process which is faster than all the others
    score[] = [0, 0, 0];

    score[ fastest01 ] = score[ fastest01 ]+1;
    score[ fastest02 ] = score[ fastest02 ]+1;
    score[ fastest12 ] = score[ fastest12 ]+1;

    winner = max( score );

    if( count[0] == winner )
        decision = input[0]
    else if( count[1] == winner )
        decision = input[1];
    else // count[2] == winner
        decision = input[2];
}

faster( i, j, id ){
    rij = R[i,j];
}
```
ri = R[i];
rj = R[j];

if ( rij == -1 ){
   // neither of i or j yet started, I am faster than both
   return id;
}
else{
   if( ri == -1 ){
      // i did not yet start, hence j must be faster
      return j;
   }
   if( rj == -1 ){
      // j did not yet start, hence i must be faster
      return i;
   }
   if( rij == i ){
      // value written by j was overridden by i
      return j;
   }
   else{ // rj == j
      return i;
   }
}

3 Analyzing a protocol

A lousy programmer wanted to solve consensus for 2 processes and came up with a sophisticated protocol. Does the protocol really solve consensus? Why?

initialize(){
   // s is shared
   s = '?';

   // i is also shared
   i = 0;

   // the input, an array of length 2
   input[] = [random({0,1}), random({0,1})];
}

// making the decision
decide() [...] // see code below
Solution

The protocol works and achieves consensus. Let’s have a closer look at the code. The loop is never executed more than twice and we can easily get rid of it, this also eliminates the variable decisionMade. All processes will pass a*, b*, and either c1* or c2*.

If we look at a* and input[0] == input[1], then the protocol trivially reaches consensus. For us only the case where the inputs differ are interesting. Out of symmetry it is enough to show that the protocol succeeds if input[0] == 0 and input[1] == 1.

When reaching b* either both processes have read the same values or they did not.

- In the case of decision0 == decision1 one process will enter the branch with c1*, the other the branch with c2*. The one passing through c1* will change its decision, the other passing through c2* will not hence both processes end with the same decision.

- In the case of decision0 != decision1 both processes have read their input. In this case both processes pass c2*. The second one will change its decision because i.fetchAndInc() returns 1, the other one will not, hence both processes end with the same decision.

- The case where decision0 != decision1 and both processes have value != decision never happens. We prove this with an execution tree for the code between a* and b*.

![Figure 1: Execution tree for the code between a* and b*](image-url)
// making the decision
decide()
   // the id of this process, 0 or 1
   id = this.getThreadId();

   //
   // a*
   /////

   value = s;
   if( value == '?' ){
      s = input[ id ];
   }
   value = s;

   //
   // b*
   /////

   if( value != input[ id ] ){
      /////
      // c1*
      /////
      decision = value;
   }
   else{
      /////
      // c2*
      /////
      if( i.fetchAndInc() == 1 ){
         decision = input[ 1-id ];
      }
      else{
         decision = input[ id ];
      }
   }
}