CS511 – Endterm – December 8 2017 – Topic 1

Exercise 1 (*Message Passing, 20 pts*)

What is the difference between synchronous and asynchronous message passing (10 pts)? Which of these mechanisms does Erlang employ (5 pts)? Support your answer to the last question with code in Erlang (5 pts).

Answer to exercise 1

In asynchronous message passing, the send operation is non-blocking. Whereas in synchronous message passing, a send operation blocks until a corresponding receive operation is executed. Erlang employs asynchronous message passing.

A simple example that illustrates this is the expression

```
1 list_to_pid("<0.100.0>")!1, io:format("hello").
```

This immediately prints "hello".

Note: Sending a message so an non existent process simply drops it. `http://erlang.org/doc/apps/erts/communication.html#id67927`

Exercise 2 (*Message Passing in Erlang, 33 pts*)

Implement a metasearch process `meta_search` that receives search requests from clients (following the message format `{request,From,Ref,Query}`) and performs searches in Google, Bing and Yahoo on their behalf. It then reports a string back to the clients with the concatenation of the results of each search engine. The PID of the Google, Bing and Yahoo search engines may be consulted in the register (eg. `whereis(google_pid)`).

Bear in mind that since searches may take long, the metasearch process, once it receives a request, should immediately become available for receiving more requests. Hence the search itself should be delegated to another thread, spawned by `meta_search`.

Answer to exercise 2

```
1  -module(ms).
2  -compile(export_all).
3
4  meta_search() ->
5   receive
6      {request,From,Ref,Query} ->
7           spawn(?MODULE,meta_search_delegate,[From,Ref,Query]),
8           meta_search()
9  end.
10
11 meta_search_delegate(From,Ref,Query) ->
12     whereis(googlePid)!{Query,self()},
13     whereis(bingPid)!{Query,self()},
14     whereis(yahooPid)!{Query,self()},
15     receive
16         {Engine1,Result1} ->
17             receive
18                 {Engine2,Result2} ->
19                     receive
20                         {Engine3,Result3} ->
21                             From!{response,Ref,lists:concat([Result1,Result2,Result3])}
22                     end
23             end
24     end.
```

Exercise 3 (*Promela, 23 pts*)

The following solution to the first part of assignment 5, namely the CR version of the Leader Election Algorithm has a problem. Its execution does not end gracefully (i.e. without errors): it ends in a `timeout`. Here is the output:

```
$ spin cr.pml
               Node is running: 3
                            Node is running: 2
          Node is running: 1
                   Node is running: 4
                       Node is running: 5
                       Node elected as leader: 5
      timeout
```

Here is the code for the node:

```
1  proctype nnode (chan inp, out; byte mynumber)
2  {
3    byte nr;
4
5    printf("Node is running: %d\n", mynumber);
6    out ! mynumber;
7  end:
8    do
9      :: inp?nr ->
10       if
11         :: nr == mynumber ->
12            printf("Node elected as leader%d\n",mynumber )
13            break
14         :: nr > mynumber ->
15            out!nr;
16         :: nr < mynumber -> skip
17       fi;
18   od;
19 }
```

1. What output you would get if you added `printf("Node finished: %d\n", mynumber);` just after line 18 and you had the code from the assignment that initializes the whole system with 5 copies of `nnode` connected in a ring. (10 pts)

2. Describe why the program ends with a timeout rather than ending correctly, without errors. Note that you are not asked to fix the problem. (13 pts).

Answer to exercise 3

1. Only the node elected as leader prints the message, namely node 5.

2. The program ends with a timeout because the only case in the do loop that breaks out of it is when the leader receives back its own message. All others are left reading from `inp` forever.

Exercise 4 (*Verification Using Assertions, 24 pts*)

In the real world messages may be lost. The Alternating Bit Protocol (ABP) is a protocol in which the sender adds a bit to every message. The receiver:

- Acknowledges each message by sending the received bit back.
- Only accepts messages with a bit that it expected to receive. Otherwise, in our simplified setting, it just ignores the message.

If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit; otherwise it resends the message.

Here is the code for the ABP in Promela. The messages that are sent are just numbers from 0 to 7 following the sequence 1,2,3,4,5,6,7,0,1,2,....

```
#define FETCH mt = (mt+1)%8

proctype Sender(chan in, out)
{
  byte mt; /* message data */
  bit at;  /* alternation bit transmitted */
  bit ar;  /* alternation bit received */

  FETCH; /* get a new message */
  printf("Sent %d with bit %d ",mt,at);
  out!data(mt,at); /* ...and send it */

  do
    ::in?ack(ar) -> /* await response */
       if
         ::(ar == at) -> /* successful transmission */
            FETCH; /* get a new message */
            printf("Sent %d with bit %d ",mt,at);
            at=1-at /* toggle alternating bit */
         ::else -> /* there was a send error */
            skip /* don't fetch a new msg. */
       fi;
       out!data(mt,at)
  od
}


proctype Receiver(chan in, out)
{
  byte mr;      /* message data received */
  bit ar;       /* alternation bit received */
  bit last_ar=1;  /* ar of last error-free msg */

  do
    ::in?data(mr,ar) -> /* send response */
       out!ack(ar);
       if
         ::(ar == last_ar) -> /* bit is not alternating */
            skip /* ...don't accept */
         ::(ar != last_ar) -> /* bit is alternating, correct message */
            printf("Got message %d with bit %d\n",mr,ar);
            last_ar=ar; /* store alternating bit */
       fi
  od
}
```

Here is an excerpt of the output:

```
$ spin abp.pml
          Sent 1 with bit 0              Got message 1 with bit 0
```

```
3           Sent 2 with bit 1               Got message 2 with bit 1
4           Sent 3 with bit 0               Got message 3 with bit 0
5           Sent 4 with bit 1               Got message 4 with bit 1
6           Sent 5 with bit 0               Got message 5 with bit 0
7           Sent 6 with bit 1               Got message 6 with bit 1
8           Sent 7 with bit 0               Got message 7 with bit 0
9           Sent 0 with bit 1               Got message 0 with bit 1
10          Sent 1 with bit 0               Got message 1 with bit 0
11          Sent 2 with bit 1               Got message 2 with bit 1
12          Sent 3 with bit 0               Got message 3 with bit 0 ^C
```

Add an assertion to ensure that the receiver only accepts the correct numbers. Hint: make sure that when it accepts a number, it should be the next one in the sequence. For that you might need an additional variable to "remember" the last accepted message.

Answer to exercise 4

```
1  proctype Receiver(chan in, out)
2  {
3    byte mr; /* message data received */
4    byte last_mr; /* mr of last error-free msg */
5    bit ar; /* alternation bit received */
6    bit last_ar; /* ar of last error-free msg */
7
8    do
9      ::in?data(mr,ar) -> /* send response */
10       out!ack(ar);
11       if
12         ::(ar == last_ar) -> /* bit is not alternating */
13            skip /* ...don't accept */
14         ::(ar != last_ar) -> /* bit is alternating */
15            assert(mr==(last_mr+1)%MAX); /* correct message */
16            last_ar=ar; /* store alternating bit */
17            last_mr=mr /* save last message */
18       fi
19   od
20 }
```

This page is intentionally left blank. You may use it for writing your answers.