CS3211 Parallel and Concurrent Programming – Week 10 tutorial

1. Design a message-passing protocol which allows a producer process communicating with a consumer process by asynchronous messaging to send only a bounded number of messages, N, before it is blocked waiting for the consumer to receive a message. Construct process equations. Your construction should avoid overflow of the message queues.

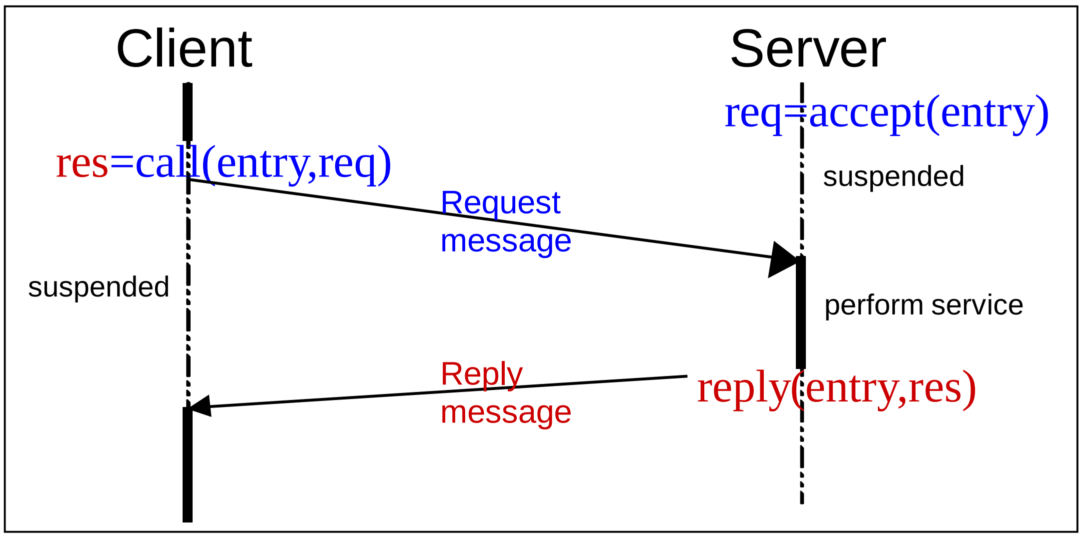
Answer:

```
// the idea here is to send a set of N tokens to
// the producer. before sending the producer must get a token
// the consumer returns tokens in response to message receipt

// Asynchronous message passing port
//(turn off "Display warning messages")

const N = 3
set   M = {msg}
set   S = {[M],[M][M]}

PORT            //empty state, only send permitted
  = (send[x:M]->PORT[x]),
PORT[h:M]       //one message queued to port
  = (send[x:M]->PORT[x][h]
    |receive[h]->PORT
    ),
PORT[t:S][h:M]  //two or more  messages queued to port
   = (send[x:M]->PORT[x][t][h]
     |receive[h]->PORT[t]
     ).

PRODUCER = (empty.receive.token -> dest.send.msg -> PRODUCER).

CONSUMER = SENDBUF[N],
SENDBUF[i:1..N] = (empty.send.token -> if (i==1) then
                                         CONTINUE else SENDBUF[i-1]),
CONTINUE = (dest.receive.msg -> empty.send.token -> CONTINUE).

||PRODCONS = (PRODUCER || CONSUMER || empty:PORT || dest: PORT)
             /{empty.[i:{send,receive}].token/empty[i].msg}.
```

2. In a rendezvous – does the server need to know the identity of the client processes? Does the client need to know the identity of the server process? In class we showed the Sequence Diagram for the case where the server process accepts after entry by the client. Try to draw the sequence diagram where the server tries to execute accept statement, prior to the client calling its entry.

Answer:

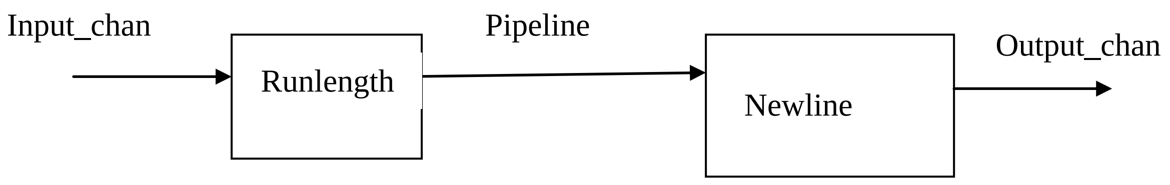
The client needs to know the identity of the server, but the server does not need to know the identity of the clients. Rendezvous is suitable for implementing servers which export their services to all possible clients.



3. [Conway's problem] The input to Conway's problem is a sequence of characters sent by an environment to an input channel. The output is the same sequence sent on another channel after performing two transformations (i) runs of $1 < n < 10$ occurrences of the same character x are replaced by the digit n and character x (ii) a newline character is appended following every Kth character of the transformed sequence produced by step (i).

Encode the problem using channels. There should be 2 processes Runlength and Newline taking care of steps (i) and (ii), and then communicate via a channel called Pipeline.

This exercise should expose you to the concept of pipelined computation which allows for tremendous parallelism, and is widely deployed in computing for extracting parallelism



```
const NEWLINE = 10

const K = 5

range M = 1 … 128

range N = 1 … 10


Runlength = input.receive[z:M] -> Runlength[z][1].

Runlength[x:M][n:N] = (when (n == 10)  Pipeline.send[n] -> Pipeline.send[x] -> Runlength
                        | when (n <10)  input.receive[z:M] -> Runlength[x][n][z]).

Runlength[x:M][n:N][z:M] =
        (when (x == z)  Runlength[x][n+1]
        | when (x != z)
              ( when (n ==1)  Pipeline.send[x] -> Runlength[z][1]
              | when (n >1)  Pipeline.send[n] -> Pipeline.send[x] -> Runlength[z][1]))).


Newline = Newline[0].

Newline[i:M] = (when (i == K)  output.send[NEWLINE] -> Newline[0]
                | when (i < K)  Pipeline.receive[y:M] -> output.send[y] ->Newline[i+1]) .


|| System = (Runlength || Newline) /{Pipeline/Pipeline.{send, receive}}
```

In Promela

```
#define N 9
#define K 4

chan inC, pipe, outC = [0]  of { byte };


active proctype Runlength(){
       byte prev, char, count = 0;


     do
     ::   inC?prev  ->
          if
          ::  (char == prev) && (count < N – 1) -> count++;
          :: else ->
              if
               ::  count >  0  ->  pipe!count+1;  count = 0;
               ::  else
              fi
              pipe! Prev;   prev = char;
          fi
     od
}


active proctype  Newline(){
          byte char, count = 0;

          do
               ::  pipe? char;  outC!char;  count++;
                   If
                      :: count >= K ->   outC! `\n’;   count = 0;
                      :: else
                   fi
           od
}
```

4. (a) Do SIMD machines need to provide synchronization primitive? Why or why not.
(b) Most SIMD machines use custom CPUs, whereas most MIMD machines use commercial CPUs. Explain why this is a common practice.

Answer:

(a) SIMD machines do not need to provide explicit synchronization primitives as SIMD supports implicit synchronization. In SIMD all the PEs run in lock step and hence it is possible to make sure statically/compile time that no two PEs try to write the same memory location or read and write the same memory location in one clock cycle.

(b) SIMD requires very simple CPU, whereas commercial CPUs are complex.

---

5. Earlier we have discussed the dining philosopher's problem where each philosopher wants to pick up the fork on her left and then right, and only then gets to eat. After eating the forks are put back. We have earlier discussed its encoding using wait/signalling (or notification). Now try to encode the problem using channels. The philosophers and the forks should each be a process, and the processes should communicate via dedicated channels. Argue that your solution avoids a deadlock. Try showing your solution using say 5 philosophers and 5 forks.

```
// Overall configuration:
// FORK(0) - chan[0] - PHIL(0) - chan[1] - FORK(1)
// FORK(1) - chan[1] - PHIL(1) - chan[2] - FORK(2)
// FORK(2) - cahn[2] - PHIL(2) - chan[3] - FORK(3)
// FORK(3) - chan[3] - PHIL(3) - chan[4] - FORK(4)
// FORK(4) - chan[4] - PHIL(4) - chan[0] - FORK(0)

// chan[i] is a channel for FORK(i).
// FORK(i) is shared between PHIL(i) and PHIL((i-1+N)%N).
// For that matter, chan[i] is shared between PHIL(i) and PHIL((i-1+N)%N).


// The WAITER_PROC process prevents deadlock. This waiter does not allow a
// philosopher to get a fork if there is only one fork left.

const N = 5

// PHIL(I) can send "get" and "put" messages to chan[i] and chan[(I+1)%N].

PHIL(I=1) =
   (chan[I].send.get[I] -> // send the "get" message and phil ID I to chan[I]
    chan[(I+1)%N].send.get[I] -> chan[I].send.put[I] -> chan[(I+1)%N].send.put[I]
-> PHIL
   ).

FORK(I=1) =
   (chan[I].receive.get[I] -> chan[I].receive.put[I] -> FORK
   |chan[I].receive.get[(I-1+N)%N] -> chan[I].receive.put[(I-1+N)%N] -> FORK).

//** THIS PROCESS IS REFINED TO PREVENT DEADLOCK

WAITER_PROC = WAITER[N],
// r: remaining forks

WAITER[r:0..N] =
    // if r > 1, anything is possible.
    (when (r > 1)  chan[i:0..N-1].send.get[i] -> WAITER[r-1]
    |when (r > 1)  chan[i:0..N-1].send.get[(i-1+N)%N] -> WAITER[r-1]
    |when (r > 1)  chan[i:0..N-1].send.put[i] -> WAITER[r+1]
    |when (r > 1)  chan[i:0..N-1].send.put[(i-1+N)%N] -> WAITER[r+1]

    // if r == 1, then only the following can be true;
    // if only one fork left, one of phil should put down his fork if possible.
    // Otherwise, only a phil who already is holding one fork can obtain a fork.

    |when (r == 1) chan[i:0..N-1].send.put[i] -> WAITER[r+1]
    |when (r == 1) chan[i:0..N-1].send.put[(i-1+N)%N] -> WAITER[r+1]
    |when (r == 1) chan[i:0..N-1].send.get[(i-1+N)%N] -> WAITER[r-1]

    // if r == 0, forks should be put down
    |when (r == 0) chan[i:0..N-1].send.put[i] -> WAITER[r+1]
    |when (r == 0) chan[i:0..N-1].send.put[(i-1+N)%N] -> WAITER[r+1]
    ).

||PHILS = forall [i:0..N-1] PHIL(i).
||FORKS = forall [i:0..N-1] FORK(i).
||SYS = (PHILS || FORKS || WAITER_PROC)
        /{chan[i:0..N-1].get/chan[i].{send,receive}.get,
          chan[i:0..N-1].put/chan[i].{send,receive}.put}.
```

More from Wikipedia

http://en.wikipedia.org/wiki/Dining_philosophers_problem

Another relatively simple solution is achieved by introducing a waiter at the table. Philosophers must ask his permission before taking up any forks. Because the waiter is aware of how many forks are in use, he is able to arbitrate and prevent deadlock. When four of the forks are in use, the next philosopher to request one has to wait for the waiter's permission, which is not given until a fork has been released. The logic is kept simple by specifying that philosophers always seek to pick up their left hand fork before their right hand fork (or vice versa). The waiter acts as a semaphore, a concept introduced by Dijkstra in 1965.

To illustrate how this works, consider that the philosophers are labelled clockwise from A to E. If A and C are eating, four forks are in use. B sits between A and C so has neither fork available, whereas D and E have one unused fork between them. Suppose D wants to eat. When he wants to take up the fifth fork, deadlock becomes likely. If instead he asks the waiter and is told to wait, we can be sure that next time two forks are released there will certainly be at least one philosopher who could successfully request a pair of forks. Therefore deadlock cannot happen.