TDT4165 PROGRAMMING LANGUAGES

Fall 2012

Exercise 7
Message-Passing Concurrency & Threads

Before you start:

- Read chapter 5.1, 5.2 and chapter 4.1-4.4 (threads) in *CTMC*. Be sure that you understand the syntax and semantics of message-passing concurrency.
- You may use the `NewPortObject` and `NewPortObject2` methods on page 351 throughout the exercise.

Task 1: A Math Agent

In this task we are going to make a simple math agent that will perform basic arithmetic operations upon request from a client. The math agent listens for incoming requests on a port and behaves according to the specifications below:

a) It should have an initial value of 0

b) It should create a new port object that will be used by clients to send messages to the math agent:

Messages received by the math agents are records of the following form:

1. `add(Number)` - add the number to the current value of the math agent
2. `sub(Number)` - subtract the number from the current value of the math agent
3. `mult(Number)` - multiplies the number to the current value of the math agent
4. `divide(Number)` - divides the number to the current value of the math agent
5. `get(Number)` - sets `Number` to the current value of the math agent

The following code should be able to run your function:

```
declare
fun {MathAgent}
```

```
   % your code
end

MathPObj = {MathAgent}
{Send MathPObj add(3)}
{Send MathPObj sub(7)}
{Browse {Send MathPObj get($)}} % displays ~4

{Send MathPObj add(27)}
{Browse {Send MathPObj get($)}} % displays 23

MathPObj2 = {MathAgent}
{Send MathPObj2 mult(27)}
{Browse {Send MathPObj2 get($)}} % displays 0
```

Task 2: A Dating Service

You are going to implement a dating service that matches potential partners. When registering at your dating service, customers must provide some information; name, sex, hair description, the wanted partners sex and the wanted partners hair description.

You are provided with a dating service client `NewClient` (see the code below). This function registers the information and preferences of new clients. It then sends a seeking request to the `NewDatingService`, and waits for a reply. Be sure you understand the provided code.

The `NewDatingService` receives `seeking` requests from the client. The requests should be formed as
`seeking(MySex MeDesc OtherSex OtherDesc ResponsePort)`.

The `NewDatingService` should keep a `state(women:Women men:Men)` of all registered customers. The initial value of this state should be `state(women:nil men:nil)`.

The `NewDatingService` should process the `seeking` request by adding the customer to the state, and search the current state for matches.

After the `seeking` request is processed, the `NewDatingService` will send responses to the clients *if* any compatible persons are found. A `list(CompatiblePeople)` should be sent to the waiting client, and a
`match(MySex MeDesc OtherSex OtherDesc ResponsePort)` should be sent to the clients that matched the waiting clients request.

For example,

```
Alice = {NewClient 'Alice' female blond male dark}
Bob = {NewClient 'Bob' male dark female blond}
```

will result in these messages sent:

- Alice registers - the 'database' is empty, no matches are found and no response is issued.
- Bob registers - Bob matches Alices request. A message is sent to every waiting client that matches Bobs characteristics. In this case only Alice. The Alice client will then receive a `match(male dark female blond <Port>)` reply. Bob will also receive a reply because there is a match for him in the database. Bob will receive `list([seeking(female blond male dark <Port>)])` which is Alice.

Make sure you understand the objectives of the task before you start working on it. It might be a good idea to consult with student assistants for advise. The important part here is to understand that a new client (with a response port) is made every time a new person registers (calls `NewCLient`). All these clients sends messages to the same `NewDatingService` and waits for responses. The `NewDatingService` has control over all registered users, and tries to find matches each time a new client registers. Responses must be sent to the new client, and to the existing matching clients each time a match is found. It is not important to make your function compatible with these specific code examples, but it is important to make a function that acts according to this description. You should implement a working dating service.

```
declare
% Messages sent to the Dating Service
%   seeking(MySex MeDescription OtherSex OtherDescription ResponsePort)
% Messages sent by the Dating service on ResponsePort
%   list(CompatiblePeople)
%   match(CompatiblePerson)

%% The port object factory
fun {NewDatingService}
   % Fill in the functionality of the datingservice
end

DatingService = {NewDatingService}

fun {NewClient Name MySex MyDesc OtherSex OtherDesc}
   RP = {NewPortObject2
         proc {$ Msg}
            {Browse Name#' got '#Msg}
         end}
in
   {Send DatingService seeking(MySex MyDesc OtherSex OtherDesc RP)}
   RP
end

Alice = {NewClient 'Alice' female blond male dark}
```

```
Bob = {NewClient 'Bob' male dark female blond}
Candice = {NewClient 'Candice' female blond male dark}
Dale = {NewClient 'Dale' male dark female blond}
Elise = {NewClient 'Elise' female brunette male blond}
Fred = {NewClient 'Fred' male blond female brunette}
{Browse done}
```

Task 3: RMI

a) Read section 5.3.1 to understand Remote Method Invocation functionality.

b) Create a server that handles the desired functionality of the client below. The functionality should match the math agent of Task 1. Why is it better to do it this way than have an agent like in Task 1?

```
declare
proc {ServerProc Msg}
   {Browse servergot(Msg)}
   % fill in server functionality
end

Server = {NewPortObject2 ServerProc}

proc {ClientProc Msg}
   {Browse clientgot(Msg)}
   case Msg
   of work(?Y) then Y1 Y2 Y3 Y4 in
      {Send Server add(10 10 Y1)} % prints servergot(add(10 10 20))
      {Wait Y1}
      {Send Server sub(10 10 Y2)} % prints servergot(sub(10 10 0))
      {Wait Y2}
      {Send Server mult(10 10 Y3)} % prints servergot(mult(10 10 100))
      {Wait Y3}
      {Send Server divide(10 10 Y4)} % prints servergot(divide(10 10 1))
      {Wait Y4}
      Y=Y1+Y2+Y3+Y4
   end
end

Client = {NewPortObject2 ClientProc}

{Browse {Send Client work($)}} % prints 121
```

Task 4: RMI with callback

a) What is the advantage of adding a callback to an RMI function?

b) Describe the different ways of implementing a callback to an RMI function.

Task 5: Threads I

a) Write a function {`IntLister Start End`} that prints a new int every second (Hint: research an Oz function named Delay). It should start by printing the Start number, and print every whole number until it reaches the End number. It should print nil when the End number is reached. For example,

```
local X in
   thread X = {IntLister 0 10} end
   {Browse X}
end
```

should print `0|1|2|3|4|5|6|7|8|9|nil` It is important that the function has delay between each number it prints.

b) Write a function Sum S Stream that prints the sum of a stream so far every time it receives a new number from the stream. S is the start sum, and Stream is the stream to receive numbers from. You can use the function from task a) to test this function.

```
local X in
   thread X = {IntLister 0 10} end
   thread Y = {Sum 0 X} end
   {Browse Y}
end
```

should print `0|0|1|3|6|10|15|21|28|36|45`

Task 6: Threads II

a) What does it mean that threads are interleaving?

b) How are threads scheduled in oz?