

Questions about Assignment

- **Interpreting error messages:**

Error in process <0.38.0> with exit value:

{function_clause, [{ring, waitSend, [<0.39.0>]}]}

Exit values are documented in the Erlang manual online.

This one means: died trying to call ring:waitSend with argument <0.39.0> (a Pid) because there was no matching function clause. Problem: the argument name began with a lowercase letter.

Objects simulated in C

- In C, objects can be emulated by using structs

```
typedef obj = struct {m1type *m1; m2type *m2;  
    d1type d1... }  
obj *o1 = oFactory(...init values);  
o1->m1(o1, ...);
```

- C function pointers refer only to function code; not function *closures* (code together with lookup environment) so all data needed to specialize the code has to be included in the struct
- Conceptually not hard, but requires explicit code to do it

Objects simulated in functional style

- Objects can be simulated using records that contain *function closures*
- Example: single method that returns previous state and sets new state:

```
newFoo(InitState) -> (fun(NewState) ->  
    {InitState, newFoo(NewState)}  
end).
```

```
F0 = newFoo(3).
```

```
{v1, F1} = F0(hi).
```

```
{v2, F2} = F1(6).
```

Generalizing

- Now both result and new state are functions of previous state and args.

```
newFoo(InitState) -> (fun(Args) ->  
    {f1(InitState, Args),  
    newFoo(f2(InitState, Args))}  
    end).
```

```
F0 = newFoo(3).
```

```
{V1, F1} = F0(hi).
```

```
{V2, F2} = F1(6).
```

Using a tuple for multiple methods (easily extend to record so methods have names)

```
newFoo(InitState) -> {  
  fun (Args1) ->  
    {f11(InitState, Args1),  
     newFoo(f12(InitState, Args1))}  
end,  
fun (Args2) ->  
  {f21(InitState, Args2),  
   newFoo(f22(InitState, Args2))}  
end,  
...  
}.
```

Using Processes its clearer and easier

```
newServer(InitState) ->  
  Server = spawn(server),  
  {fun(Args1) ->  
    rpc(Server, {f1, Args1}),  
    fun(Args2) ->  
    rpc(Server, {f2, Args2}),  
    ...  
  }  
end.
```

- **rpc** here is just the same **rpc** function we've seen before
- **f1** and **f2** are atoms in this context

The server

```
server(State) ->
  receive
    {From, {f1, Args1}} ->
      From ! f11(State, Args1),
      server(f12(State, Args1));
    {From, {f2, Args2}} ->
      From ! f21(State, Args2),
      server(f22(State, Args2))
  ...
end.
```

- **f11, f12, f21, and f22 are the same functions as in the purely sequential simulation**

The client

```
Server1 = newServer(SomeState),
```

```
{F1, F2} = Server,
```

```
V1 = F1(Args1),
```

```
V2 = F2(Args2), % note V2 depends on both  
Args1 and Args2 as well as SomeState
```

```
Server2 = newServer(SomeOtherState),
```

```
V3 = F1(OtherArgs1),
```

```
V4 = F2(OtherArgs2), % note V4 depends on  
OtherArgs1 and 2 and SomeOtherState but  
not on SomeState, Args1 and Args2
```

The behavior is not quite the same

- **If the sequential simulation is used from multiple processes each gets its own “fork” of the object history**
 - **Because the way it is used results in a new object every time a method is called**
- **The process simulation involves only a single object that, oh by the way, has synchronized access**
 - **A process, inherently, does only one thing at a time**

A complete, simple example

```
-module(tester).
```

```
-export([start/1]).
```

```
start(Id) ->
```

```
    Server = spawn(fun () -> serverloop(Id, []) end),  
    fun(Item) -> rpc( Server, Item ) end
```

```
.
```

```
serverloop(Id, History) ->
```

```
    receive
```

```
        { From, Something } ->
```

```
            io:format( "Server ~p: Got ~p from ~p with history ~p~n", [Id,  
                Something, From, History] ),
```

```
                From ! { self(), ok },
```

```
                serverloop(Id, [Something|History])
```

```
end.
```