Assignment 6

Erlang

Due Date: Monday, April 17th 2017, 11:59pm

Directions

We will take some points off for: code with the wrong type or wrong name, duplicate code, code with extra unnecessary cases, or code that is excessively hard to follow. You should always assume that the inputs given to each function will be well-typed, thus your code should not have extra cases for inputs that are not of the proper type. (Assume that any human inputs are error checked before they reach your code. Avoid duplicating code by using helping functions, library functions (when not prohibited in the problems). It is a good idea to check your code for these problems before submitting. Be sure to test your programs on Eustis to ensure they work on a Unix system.

This is a group assignment. Make a group of 3-4 in WebCourses and use that to turn in the assignment.

Deliverables

For this assignment, turn in the following files:

- barrier.erl
- electionserver.erl
- eventdetector.erl

For problems that require an English answer, put your solution in Solution.pdf. For coding problems, put your solution in the appropriate Erlang source file (named with ".erl" suffix).

Testing Code

The testing code for Erlang is given in the testing.erl file supplied with this homework's zip file. Note that to run our tests, all your modules involved must first be compiled. Then to execute our tests in a module named m_tests, type m_tests:main(). to the Erlang prompt.

Problem 1 (25 pts)

In Erlang, write a barrier synchronization server in a module `barrier`. In barrier synchronization, a group of processes wait until all of them are done executing up to a certain point (the barrier). You will write a function `start/1`, which takes a positive integer, which is the size of the group of processes, and create a barrier synchronization server, returning its process id. The barrier synchronization server tracks in its state the number of processes that are still running (are not yet done), and the process ids of all processes that have reached the barrier. The server responds to messages of the following forms:

- $\{\mathrm{Pid}, \mathrm{done}\}$, where Pid is the process id of the sender. The server responds sending a message to Pid of the form $\{\mathrm{SPid}, \mathbf{ok}\}$, where SPid is the server's own process id. What it does next depends on whether Pid was the last process in the group to finish. If Pid is the last process in the group to be done (i.e., if there are no other running processes), then Pid and all the processes that have previously sent such a done message are sent a message of the form $\{\mathrm{SPid}, \mathrm{continue}\}$, which lets them continue past the barrier. If there are other running processes, then the server just remembers Pi in the list of processes that have reached the barrier (and are thus waiting).
- $\{\mathrm{Pid}, \mathrm{how\_many\_running}\}$, where Pid is the process id of the sender. The server responds by sending a message to Pid of the form $\{\mathrm{SPid}, \mathrm{number\_running\_is}, \mathrm{Running}\}$, where SPid is the server's own process id and Running is the number of processes in the group that have no yet reached the barrier. The server continues with an unchanged state.

You can assume that each process in the group only sends a done message to the server once. (But despite this, the server does not "reset" or start over, but keeps running once all the processes in the group are done.)

To run our tests, run `barrier_tests:main()`.

Problem 2 (25 pts)

In this problem you will write an election server and two client functions. The three functions you are to implement are the following.

```
-spec start() -> pid().
```

The `start/0` function which creates a new election server and returns its process id.

```
-spec vote(ES::pid(), Candidate::atom()) -> ok.
```

The `vote/2` function takes as arguments the process id of an election server, and a candidate's name (an atom). By sending messages to the election server, this function casts a single vote the candidate. After the server has received the vote, this function returns the atom ok to the caller.

```
-spec results(ES::pid()) -> [{atom(), non_neg_integer()}].
```

The `results/1` function takes the election server's process id as an argument. It returns a list of pairs of the form $\{\mathrm{Candidate}, \mathrm{Vote}\}$, where `Candidate` is a candidate's name, and `Vote` is the total number of votes cast for the candidate. The returned list sorted in the order given by `lists:sort/1`, i.e., in non-decreasing order by the candidate's name. This function does not change the state of the server.

To run our tests, run `electionserver_tests:main()`.

Problem 3 (30 pts)

In Erlang, write a module `eventdetector`, which has a function `start/2` that takes two arguments: `InitialState`, and `TransitionFun`. For each type of state, S, the `InitialState` has type S and the `TransitionFun` has type `fun((S,atom()) -> {S,atom()});` that is, it is a function that takes a state and an atom and returns a pair of a state and an atom. A call to start/2 makes a server process that tracks a state and a list of observers. The observers are remembered by remembering their process ids. An event detector server responds to the following messages.

- $\{\mathrm{Pid}, \mathrm{add\_me}\}$, which sends the message $\{\mathrm{added}\}$ (i.e., a singleton tuple containing the atom added) to the process id Pid, and then adds the process id Pid to the head of the list of observers that the event detector is remembering. (The remembered state is unchanged.)
- $\{\mathrm{Pid}, \mathrm{add\_yourself\_to}, \mathrm{EDPid}\}$, which sends the message $\{\mathrm{self}(), \mathrm{add\_me}\}$ to EDPid, and then waits for the server EDPid to respond with the message $\{\mathrm{added}\}$, which it sends to Pid. (The remembered state and list of observers of this event detector itself are unchanged.)
- $\{\mathrm{Pid}, \mathrm{state\_value}\}$, which sends the current state of the server to Pid in a message of the form $\{\mathrm{value\_is}, \mathrm{State}\}$, where State is the event detector's remembered state. (The remembered state and list of observers of this event detector are unchanged.)
- an atom, which causes the event detector to call TransitionFun on the current state and the atom received, which yields a pair $\{\mathrm{NewState}, \mathrm{Event}\}$. If Event is the atom none, then no observers are sent messages; otherwise, if Event is any other atom (other than none), then Event is sent to all observers in the server's list of observers (in the order of that list). After handling any needed notification of the observer, then the event detector continues with the state NewState and an unchanged list of observers.

To run our tests execute `eventdetector_tests:main()`.