Computer Science 483/580
Concurrent Programming
Midterm Exam 2 – Sample

Your name ____

There are 6 pages to this exam printed front and back. Please make sure that you have all the pages now.

This exam is **open book and open notes.** You may use a laptop computer to refer to an electronic copy of the textbooks and to your notes. You ***must not*** use your computer to ***access the network or use any compilers, interpreters, etc.***; if possible turn off the network interface. Other electronic devices – e.g. cellphones – may not be used.

Honor statement: I have followed the above instructions regarding use of electronic devices while taking this exam.

Signature ______________________________________ Date _____________

1. (15 pts) Give an example of an Erlang **`receive`** expression that will process the message `{one, 1}` if it is present in the process's mailbox and will otherwise process the first message in the mailbox.

Solution approach: review the operation of receive and determine whether the nesting of the loops (over clauses of the receive and messages in the mailbox) allows this to be done in a single receive or whether a receive with a 0 timeout followed by another receive is required.

2. Terminology in the area of concurrent programming is often inconsistent and confusing. For example, another term for threads, such as those found in Java, is *lightweight processes.* At the same time we have claimed that Erlang processes are much lighter weight than Java threads.

a) (10 pts) Explain in what sense is it reasonable to describe Java threads as lightweight processes.

b) (10 pts) Explain why, nevertheless, Erlang processes are lighter weight than Java threads.

3. a) (10 pts) Explain why, in Erlang,

```
Pid = spawn(F),
link(Pid).
```

is not the same as

```
Pid = spawn_link(F).
```

b) (15 pts) In the ring assignment, one way to implement the message forwarding processes was with a loop function like this:

```
loop (Successor) ->
     receive
          M -> Successor ! M,
          loop (Successor)
     end.
```

The message forwarding processes are created by calling `spawn(ring, loop, [Successor]).` A problem with this approach is that after all the messages of a run have been forwarded, the forwarding processes hang around waiting for messages that will never be received. How could you use Erlang's process linking mechanism to kill off all of the forwarding processes with a single function call once the last message is received by the last process? Describe what needs to be done when the processes are created and what needs to be done when the last message has been received.

4. Below is an implementation of a simple generic server that is customized by passing a single function and an initial state to `server:newServer`.

```
-module(server).
-export([newServer/2, rpc/2]).
% Implementation of a simple generic server.
% The server is specialized by the function, F, passed to
% newServer

% Create a new server with F as the function for
% calculating the response and the new state
newServer(F, InitState) ->
   spawn(server, loop, [F, InitState]).

% the server loop
loop(F, State) ->
   receive
      {Requestor, M} ->
           {NewState, Response} = F(State, M),
           Requestor ! {self(), Response},
           loop (F, NewState)
   end.

% the usual rpc function
rpc(Server, Msg) ->
   Server ! {self(), Msg},
   receive
      {Server, Response} -> Response
   end.
```

a) (20 pts) Define an Erlang function, `fflogic`, that could be passed to `server:newServer` along with a value indicating the initial state to simulate a flip-flop as follows:

a. the flip-flop has two states represented by atoms: `off` and `on`.

b. to create a flip-flop server the client uses `FFServer = server:newServer(fflogic, off)`.

c. to learn the current state of the flip-flop the client uses `FFState = server:rpc(FFServer, getState)`

d. to toggle the state of the flip-flop (from `off` to `on` or vice-versa) the client uses `server:rpc(FFServer, toggle)`. The toggle operation returns the state of the flip-flop *before* toggling it.

Note: this problem requires you to follow the Erlang style of taking advantage of existing generic concurrent code, the server module on the previous page, and specializing it with a function that implements the specific behavior required. Do NOT reproduce the functionality of the generic server code in your answer. MAKE SURE THAT YOU UNDERSTAND THIS POINT! Not only will you get credit for the problem you will also find it much easier to solve the problem if you concentrate on what is required instead of on doing what is not required.

Solution:

Before worrying too much about what fflogic is supposed to do let's first figure out the general shape of of the function: what are its arguments and return values? We have to inspect the server module to figure this out: we see that the first argument is the State of the server and the second is the message that was received. The result has to be a tuple consisting of the NewState and the response to send to the client. From the specification of the problem, we see that the state is always either on or off.

```
fflogic(on, getState) -> {on, on};
fflogic(on, toggle) -> {off, on};
fflogic(off, getState) -> {off, off};
fflogic(off, toggle) -> {on, off}.
```

b) (20 pts) Define an Erlang function, `timerLogic,` that could be passed to `server:newServer` along with a value indicating the initial state to simulate a timer as follows:

a. the timer has two visible states, `off` and `on.` Additional states or state information may be used internally if needed.

b. to create a timer server the client uses `TServer = server:newServer(timerLogic, off).`

c. to learn the current state of the timer the client uses `TState = server:rpc(TServer, getState).` The client must be able to get an immediate answer from the server at any time. The answer is either `off` or `on`.

d. to turn `on` the timer, the client uses `server:rpc(TServer, start).` The timer is to remain `on` for 1000ms after the last start message it receives. For example, if at time 500ms a start message is received the timer will remain on until 1500ms, unless another start message is received in which case it will remain on until 1000ms after that message. The return value of start is immaterial.

e. You may use the `stimer` module (Section 12.4 of the Erlang book) and reproduced below:

```
-module(stimer).
-export ([start/2, cancel/1]).
start(Time,Fun) -> spawn(fun() -> timer(Time, Fun) end).
cancel(Pid) -> Pid ! cancel.
timer(Time, Fun) ->
  receive
    cancel -> void
  after Time ->
    Fun ()
  end.
```

Note: When I say in a problem that you may use something I mean you may assume that it already exists and you can take advantage of it in your solution to the given problem (and I advise that you do!).

Begin by asking yourself “what is required to solve this problem that is not already available from stimer?” and, “how could I use stimer to implement part of what is required?”

Solution on the next page.

```
timerLogic(off, getState) -> {off, off};
timerLogic({on, TimerPid}, getState) ->
   {{on, TimerPid}, on};
timerLogic(off, start) ->
   timerStartOrRestart();
timerLogic({on, OldTimerPid}, start) ->
   stimer:cancel(OldTimerPid),
   timerStartOrRestart();
timerLogic({on, _}, timeExpired) ->
   {off, dontcare};
% the following should never occur but is include
% for completeness
timerLogic(off, timeExpired) -> {off, dontcare}.

timerStartOrRestart() ->
   Self = self(),
   TimerPid =
      stimer:start(
         1000,
         fun () -> server:rpc(Self, timeExpired) end
      ),
   {{on,TimerPid}, dontcare}.
```

Note: the Pid of the running stimer needs to be part of the state, when on, so that the stimer can be cancelled if a start message is received while it is running. Also notice how the timeExpired message handled by the timerLogic function is sent from the process created by stimer:timer and not by the client process that uses the timerLogic server. And finally, notice that in the call to server:rpc from the fun argument to stimer:timer, Self and not self() is used as the server process id. Why?

And truly finally, notice that the relatively complicated code in timerStartOrRestart might have been duplicated for the timerLogic(off, start) case and the timerLogic({on, OldTimerPid}, start) case but in line with the principle of “Don't Repeat Yourself” it has been extracted into a function called from both locations.