Processes and Messages

Mark Greenstreet

CpSc 418 – Jan. 11, 2016

Outline:

Objectives

- Introduce Erlang's features for concurrency and parallelism
  - Spawning processes.
  - Sending and receiving messages.
- The source code for the examples in this lecture is available here: procs.erl.

A Few Announcements

- Mini Assignment revised deadline: Wednesday, Jan. 13, 10am.
- Homework 2 will go out Jan. 18; due on Feb. 3 (early bird Feb. 1)
- Lecture note bug-bounty protocol
  - Please post (alleged) bugs to piazza.
  - Only announce them in lecture if they are blocking comprehension of the material.

Processes – Overview

- The built-in function spawn creates a new process.
- Each process has a process-id, pid.
  - The built-in function self() returns the pid of the calling process.
  - `spawn` returns the pid of the process that it creates.
  - The simplest form is `spawn(`*Fun*`)`.
    - ★ A new process is created.
    - ★ The function *Fun* is invoked with no arguments in that process.
    - ★ When *Fun* returns, the process terminates.

Processes – a friendly example

```
hello(N)->
    [     spawn(fun() -> io:format(
            "hello world from process ~p.  I = ~p~n",
            [self(), I])
          end)
       || I <- lists:seq(1,N)
    ].
```

Processes – a friendly example

```
hello(N)->
    [     spawn(fun() -> io:format(
            "hello world from process ~p.  I = ~p~n",
            [self(), I])
          end)
       || I <- lists:seq(1,N)
    ].
```

- Let's run it:

```
1> c(procs).
{ok,procs}
2> procs:hello(3).
hello from process 1
hello from process 2
hello from process 3
[<0.40.0>,<0.41.0>,<0.42.0>]
```

Sending Messages

- The “hello world” example was obligatory, but the processes don’t solve any problem (other than making us all feel welcome).
  - To solve tasks in parallel, the processes need to communicate.
  - In Erlang, communication is done with messages.
- Sending a message: `Pid ! Expr`.
  - `Expr` is evaluated, and the result is sent to process `Pid`.
  - The value of a send expression is `Expr`.
  - Evaluation “succeeds” even if `Pid` is no longer a running process.
  - Message passing is asynchronous: the sending process can continue its execution before the receiver gets the message.

Receiving Messages (short version)

```
receive
    Pattern1 -> Expr1;
    Pattern2 -> Expr2;
    ...
    PatternN -> ExprN
end
```

- If there is a pending message for this process that matches one of the patterns,
  - The message is delivered, and the value of the `receive` expression is the value of the corresponding *Expr*.
  - Otherwise, the process blocks until such a message is received.

A simple example

```
3> MyPid = self().
<0.33.0>
4> spawn(fun() -> MyPid ! "hello world" end).
<0.45.0>
5> receive Msg1 -> Msg1 end.
"hello, world"
```

Adding two numbers using processes and messages

- The plan:
  - We'll spawn a process in the shell for adding two numbers.
  - This child process will receive two numbers, compute the sum, and send the result back to the parent.
- The add process:

```
add_proc(PPid) ->
   receive
      A -> receive
         B ->
            PPid ! A+B
      end
   end.

adder() ->
   MyPid = self(),
   spawn(fun() add_proc(MyPid) end).
```

Adder Demo

```
6> Apid = procs:adder().
<0.44.0>
7> Apid ! 2.
2
8> Apid ! 3.
3
9> receive Sum -> Sum end.
5
```

Reactive Processes and Tail Recursion

- Often, we want processes that do more than add two numbers together.
- We want processes that wait, receive a message, process the message, and then wait for the next message.
- In Erlang, we do this with recursive functions for the child process:

```
acc_proc(Tally) ->
   receive
      N when is_integer(N) ->
         acc_proc(Tally+N);
      {Pid, total} ->
         Pid ! Tally,
         acc_proc(Tally)
   end.

   accumulator() ->
      spawn(fun() -> acc_proc(0) end).
```

Accumulator Demo

```
10> BPid = procs:accumulator().
<0.53.0>
11> BPid ! 1.
1
12> BPid ! 2.
2
13> BPid ! 3.
3.
14> BPid ! {self(), total}.
{<0.33.0>, total}
15> receive T1 -> T1 end.
6
```

Accumulator Demo (continued)

```
16> BPid ! 4.
4.
17> BPid ! {self(), total}.
{<0.33.0>, total}
18> BPid ! 5.
5.
19> BPid ! 6.
6.
20> BPid ! {self(), total}.
{<0.33.0>, total}
21> receive T2 -> T2 end.
10
22> receive T3 -> T3 end.
21
```

Message Ordering

- Given two processes, *Proc1* and *Proc2*, messages sent from *Proc1* to *Proc2* are received at *Proc2* in the order in which they were sent.
- Message delivery is reliable: if a process doesn't terminate, any message sent to it will eventually be delivered.
- Other than that, Erlang makes no ordering guarantees.
  - In particular, the triangle inequality is not guaranteed.
  - For example, process *Proc1* can send message *M1* to process *Proc2* and after that send message *M2* to *Proc3*.
  - Process *Proc3* can receive the message *M2*, and then send message *M3* to process *Proc2*.
  - Process *Proc2* can receive messages *M1* and *M3* in either order.
  - Draw a picture to see why this is violates the spirit of the triangle inequality.

Tagging Messages

- It's a very good idea to include "tags" with messages.
- This prevents your process from receiving an unintended message:

  "Oh, I forgot that another process was going to send me that. I thought it would happen later."

- For example, my accumulator might be better if instead of just receiving an integer, it received

```
{2, add}
```

Tracing Processes

When you implement a reactive process with a tail-recursive, it can be handy to trace the execution. For

- Add an `io:format` call when entering the function and after matching each receive pattern.
- Example:

```
acc_proc(Tally) ->
    io:format("~p:  acc_proc(~b)~n", [self(), Tally]),
    receive
        N when is_integer(N) ->
            io:format("~p:  received ~b~n", [self(), N]),
            acc_proc(Tally+N);
        Msg = {Pid, total}
            io:format("~p:  received ~p~n", [self(), Msg]),
            Pid ! Tally,
            acc_proc(Tally)
    end.
```

- Try it (e.g. with the example from slide 12.
- Don't forget to delete (or comment out) such debugging output before releasing your code.

Time Outs

- If your process is waiting for a message that never arrives, e.g. because
  - You misspelled a tag for a message, or
  - The receive pattern is slightly different than the message that was sent, or
  - Something went wrong in the sending process, and it died before sending the message, or
  - You got the message ordering slightly wrong, and there's a cycle of processes waiting for each other to send something, or
  - ...
- Then your process can wait forever, your Erlang shell can hang, and it's a very unhappy time in life.
- Time-outs can handle these problems more gracefully.
  - See Time Out in *LYSE*.
  - Note: time-outs are great for debugging, they should be used with great caution elsewhere because they are sensitive to changes in hardware, changes in the scale of the system, and so on.

Summary

- Processes are easy to create in Erlang.
  - The `spawn` mechanism can be used to start other processors on the same CPU or on machines spread around the internet.
- Processes communicate through messages
  - Message passing is asynchronous.
  - The receiver can use patterns to select a desired message.
- Reactive processes are implemented with tail-recursive functions.
- Time-outs and print statements (e.g. `io:format`) are handy for debugging.

Preview

January 13: Reduce and Scan (simple)

Reading:	Lin & Snyder, chapter 5, pp. 112–125
Mini-Assignment:	Mini-Assignment 2 due **10:00am**

January 15: Reduce and Scan (generalize)

Homework:	Homework 1 deadline for early-bird bonus (11:59pm)

January 18: Architecture Review

Reading:	Pacheco, Chapter 2, Sections 2.1 and 2.2.
Homework:	Homework 1 due **11:59pm**
	Homework 2 goes out – parallel programming with Erlang

January 20: Shared-Memory Machines

Reading:	Pacheco, Chapter 2, Section 2.3

January 22: Distributed-Memory Machines

Reading:	Pacheco, Chapter 2, Sections 2.4 and 2.5.

January 25-29: Parallel Performance

Reading:	Pacheco, Chapter 2, Section 2.6.

Review Questions

- How do you spawn a new process in Erlang?
- What guarantees does Erlang provide (or not) for message ordering?
- Give an example of using patterns to select messages.
- Why is it important to use a tail-recursive function for a reactive process?
  - In other words, why is it a bad idea to use a head-recursive function for a reactive process.
  - The answer isn't explicitly on the slides, but you should be able to figure it out from what we've covered.
- Modify one of the examples in this lecture to use a time-out with one or more `receive` operations. Try it and show that it works.
- Implement the message flushing described in *LYSE* to show pending messages on a time-out. Demonstrate how it works.

Table of Contents