ERLANG CONCURRENCY

Curt Clifton
Rose-Hulman Institute of Technology

SVN Update *ErlangInClass*

A deep understanding of concurrency is hardwired into our brains.

The world is parallel.

Erlang programs model how we think and interact.

“We don’t have shared memory. I have my memory. You have yours. We have two brains, one each. They are not joined together. To change your memory, I send you a message: I talk, or I wave my arms. You listen, you see, and your memory changes; however, without asking you a question or observing your response, I do not know that you have received my messages.”

CONCURRENCY IN ERLANG

- Erlang programs are made of lots of processes
- Processes send messages to each other
- Messages may or may not be received
    - Processes must explicitly communicate back if acknowledgment is needed
- Pairs of processes can be linked

ERLANG PROCESSES ARE PART OF THE LANGUAGE

- Creating and destroying processes is fast
- Sending messages is fast
- Processes behave the same on every OS
- Can have huge numbers of processes
- Processes do not share memory
- Processes interact through message passing

Q1

FIRST SOME SHELL FOO

```
1> self().
<0.30.0>
2> receive foo -> true end.
control-G here
User switch command
 --> h
  c [nn]    - connect to job
  …
  s         - start local shell
  …
  ? | h     - this message
 --> j
   1* {shell,start,[init]}
 --> s
 --> j
   1  {shell,start,[init]}
   2* {shell,start,[]}
 --> c 2
Eshell V5.6.2  (abort with ^G)
```

use original numbers here

```
1> self().
<0.35.0>
2> pid(0,30,0) ! foo.
foo
3> control-G here
User switch command
 --> j
   1  {shell,start,[init]}
   2* {shell,start,[]}
 --> c 1
enter here
3> control-G here
User switch command
 --> j
   1* {shell,start,[init]}
   2  {shell,start,[]}
 --> k 2
 --> j
   1* {shell,start,[init]}
 --> c 1
```

JUST THREE PRIMITIVES FOR CONTROLLING PROCESSES

- Spawn – creates new processes
- Send – sends a message to a running process
- Receive – processes incoming messages

SPAWN

- *spawn(module, function, args)*
  - Creates a new process and starts it by evaluating the given *function* on the given *args* list
  - Process runs until function terminates
  - Typically function is an infinite tail recursion
  - *spawn* returns the "Process ID" of the new process

Q2

SEND

- *Pid ! Message*
  - *Pid* is a process ID
  - *Message* is any Erlang value
  - Message **sending** is asynchronous
    - Sender continues immediately to next expression
    - “Non-blocking”

Result of *Pid ! Msg* is *Msg*, so you can chain: *PidA ! PidB ! Msg.*

Q3

RECEIVE

- Syntax:
```
receive
    MsgPattern1 [when Guard1] -> ExpSeq1;
    MsgPattern2 [when Guard2] -> ExpSeq2;
    …
end
```
- Incoming messages are pattern matched
  - Match found, run expression sequence
  - No match found, store message for later processing and wait for next incoming message
- Message **receive** is blocking

Q4

RECORDS IN ERLANG

Declares record type

- **-record(stu, {name, year = 1}).**

- **S1 = #stu{name = “Jerry”}.**
  #stu{name = “Jerry”, year = 1}

Creates record

- **S2 = S1#stu{year = 2}.**
  #stu{name = “Jerry”, year = 2}

Creates record from existing one

- **#stu{name = N, year = Y} = S2.**
  N.
  “Jerry”
  Y.
  2

Read records

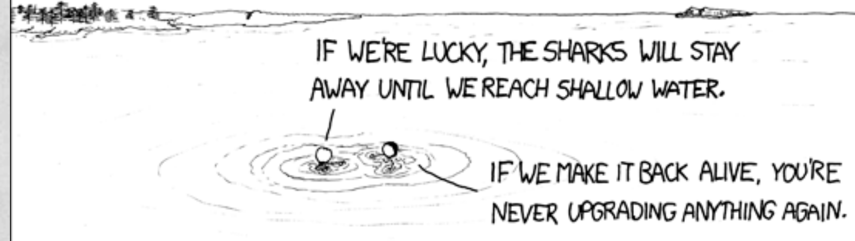
- **S2#stu.name.**
  “Jerry”

Q5

EXAMPLE: LIFTS_V1

- Open *lifts.hrl*
  - Look at record declaration
- Open *lifts_v1.erl*
  - Start in *start_car/1*
  - Look at *spawn*, *receive*, send from the shell
  - Add other messages to *car_loop*

AS A PROJECT WEARS ON, STANDARDS FOR SUCCESS SLIP LOWER AND LOWER



40% of OpenBSD installs lead to shark attacks.
It's their only standing security issue.

THE **TRUTH** ABOUT SEND AND RECEIVE

Sending Code

Pid ! {msgX}
Pid ! {msgY}

Receiving Process

Mailbox

Save Queue

Code with *receive*

```
loop() ->
  receive
    {msgY} -> doY();
    {msgZ} -> doZ();
  after 1000 -> timeout
  end,
  receive
    {msgX} -> doX(), loop();
  end.
```

MAKING THE PROCESS SEND MESSAGES BACK

- Open *lifts_v2.erl*
- Notice:
  - Receive loop now expects to know who's asking for information
  - New *car_*command functions send *self()*, then wait for message back
    - *self()* yields the Pid of the current process