

# Erlang concurrency

## Where were we?

- Finished talking about sequential Erlang
- Left with two questions
  - retry – not an issue; I mis-read the statement in the book. Erlang does *not* have retry
  - Which exceptions are caught in try...catch. Only those in the `FuncOrExpressionSequence` i.e. exactly as described and non-intuitive.

# Concurrency mechanisms - processes

- Concurrent activities are called *processes*
- Erlang processes are implemented by the language system, not the OS
  - Very lightweight even compared to Java threads
- Processes are created with `spawn/1`  
`Pid = spawn(Function)`
  - Function must be nullary (take 0 args)
- Or `spawn/3`  
`Pid = spawn(Module, Function, ArgList)`
  - Args is a list; Function must take that many args

# Concurrency mechanisms - mailboxes

- Every process has exactly one mailbox
- The mailbox contains messages. Messages simply are Erlang values.
- Messages in the mailbox are in arrival order
- Sending messages  
`Pid ! Msg`
- Note: `(Pid ! Msg) ::= Msg`
- Note also: sending is asynchronous – the sender does not wait for the receiver

# Concurrency mechanisms – selective receive

`receive`

`Pattern1 [when Guard1] -> Exprs1;`

`Pattern2 [when Guard2] -> Exprs2;`

`...`

`[after`

`Time -> ExprsT]`

`end`

## Execution of receive

- 1. For each message,  $m$ , in the mailbox, attempt to match it to each of the patterns/guards of the receive operation**
  - If a match succeeds, the result of receive is the result of evaluating the corresponding expression (the matched message is removed from the mailbox)**
- 2. Otherwise, wait and as new messages arrive test them against the patterns/guards until one succeeds (and the result comes from evaluating the corresponding expression) or the timeout occurs (and the result comes from the timeout expression).**

**(Different, equivalent, explanation in the book.)**

# Erlang receive is powerful

- Many concurrent languages allow only the first message in the mailbox to be received
  - Erlang approach allows one mailbox to emulate many (perhaps at some cost in execution time)
  - Erlang approach allows application to decide on message priority without any special language mechanism
- Asynchronous paradigm is a good match for distribution and is relatively easy to implement
- Synchronous message passing is more powerful in some ways, but is not a good match for distributed systems

# Lurking under the covers

- **Synchronization**
  - **A processes mailbox is a shared data structure, updated by more than one process**
- **Visibility**
  - **Are values that are sent to mailboxes fully visible to receiving processes on modern memory architectures**
- **These are Erlang implementation questions, rather than questions for the Erlang programmer**

# Client-server

- ***Client* sends a *request* to server**
- ***Server* sends *response* to client**
- **Server usually services many clients**
  - **How does it know where to send the response message?**
  - ***Protocol* includes client's Pid as part of the request.**  
Obtain using `self()`
    - ***A protocol* is a set of rules for communicating**
- **Oops, and a client may use many servers or itself be a server**
  - **Protocol also includes the server's Pid in the response**

## Client skeleton

```
Server ! {self(), Request},  
receive  
  {Server, Response} -> Response  
end
```

# Server skeleton

```
loop() ->
    receive {From, Request} ->
        From ! answer(self(), Request)
    end,
    loop().
```

answer() -> ...

- **Start the server with** `Server = spawn(fun loop/0)`.
- **Note:** a server, inherently, does only one thing at a time; it does synchronization
- **Note:** *stateless* server
- **Note:** tail recursion; placement of recursive call

# Software Engineering

- Don't want every use of the server to look like our client skeleton – can we write it once?
- Who starts the server?
- Answers: include code for these things in the server's module

```
start() -> spawn(fun loop/0). % result is Server  
Pid
```

```
rpc(Server, Request) ->  
  Server ! (self(), Request),  
  receive  
    (Server, Response) -> Response  
end.
```

- Note: maybe still troublesome that client deals in Server Pid rather than a Server object.

# Timeouts

- **Timeout alone (no patterns) -> sleep.**
- **Timeout 0**
  - **polling receive (don't do this);**
  - **mailbox flush**
  - **priority receive – take a particular kind of message out of order (receive processing matches each message to a pattern; timeout 0 allows to process by matching all patterns in turn to each message)**
- **Timeout infinity**
  - **Equivalent to no after clause, but allows computed infinite wait instead of static infinite wait**

# Registered Processes

- A perennial problem in client-server computing is “How do clients find the server?”
- Two possible answers:
  - Clients are told about the servers as parameters when the clients are started
    - Lots of parameters; startup order dependency
  - Clients use a *name service* to locate the servers
- Erlang answer: registered processes
  - `register(AnAtom, Pid)`
  - Can then send to the atom as if it were a Pid.
- Note: mutable global state!

# Assignment

- **Problem 2, Section 8.11; due March 12 midnight**
  - **Details about exactly what I want will be posted later this week; in the meantime you can work on**
    - **How to build a ring so that P0 sends to P1 to ... to Pn-1 to P0**
      - **Use at most 2 static code bodies (P0 may be special) for the processes in the ring**
      - **Think first about what each process needs to do – what parameters does it require? Will each process spawn its successor or will they all be spawned by a main function then glued together?**
      - **If spawning all processes from a single point, construct the ring elegantly – ideas: use map or a list comprehension or for() from page 141.**
    - **You don't have to implement in another language or write a blog 😊.**