Processos em Erlang

Paulo Ferreira
paf@dei.isep.ipp.pt

Fevereiro de 2006

Porquê?

Nas palavras de Joe Armstrong:

- ≪The world is concurrent≫
- ≪Things in the world don't share data≫
- ≪Things comunicate with messages≫

ORGC Erlang – slide 3

Ideia

- Processos totalmente independentes
- Actividades concorrentes podem ser modeladas usando processos de ≪baixo peso≫
- Muitos processos podem correr simultaneamente
- A partilha de dados é ineficiente (não pode ser feita em paralelo) e complicada (locks)

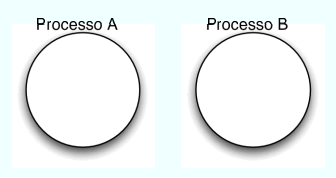
ORGC Erlang – slide 4

Prática

- Não existe partilha de dados, tudo o que fôr partilhado tem de ser passado como mensagem
- Cada processo tem um nome próprio (PID) que não se pode falsificar
- Sabendo o nome do processo pode-se mandar uma mensagem
- Não existem garantias de entrega da mensagem
- Pode-se monitorizar um processo remoto

ORGC Erlang – slide 5

Criação de Processos



- Código em PidA:
  `PidB=spawn(Módulo,Função,ListadeArgumentos)`
- A identidade do segundo processo (`PidB`) é apenas conhecida pelo Processo A.

ORGC Erlang – slide 6

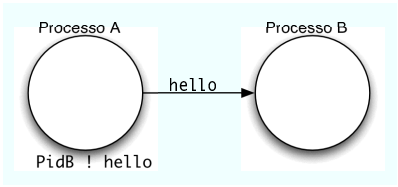
spawn

- `spawn(Módulo,Função,ListadeArgumentos)`
- Cria um novo processo, isto é uma nova linha de execução, começando na chamada da função fornecida pelos argumentos.
- Exemplo: `spawn(m,f,[1,2,3])`
  - cria um processo com a função `m:f(1,2,3)`

- A função usada em `spawn/3` deve ser exportada!
- `spawn/3` nunca falha!
- Um processo termina quando não houver mais código para executar.

Mensagens

Os processos comunicam entre si enviando e recebendo mensagens:



Envio

- As mensagens são enviadas usando ≪!≫.
  - `Pid ! Msg`
- O envio de uma mensagem nunca falha (do ponto de vista do remetente)
- `Msg` pode ser qualquer termo legal em Erlang (àtomo,tuplo ou lista)
- Normalmente é um tuplo em que um dos elementos do tuplo é um àtomo que serve de ≪etiqueta≫

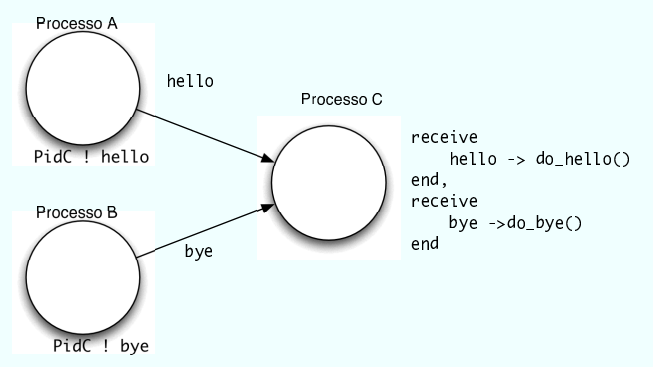
Recepção

- As mensagens são recebidas usando `receive`
- A sintaxe é similar à sintaxe do `case`:

```
receive
{Pid, hello} ->
        io:format("~p: got hello from ~p~n", [self(), Pid]),
        f();
{Pid, Msg} ->
        io:format("~p: got ~p from ~p~n", [self(), Msg, Pid]),
        g();
quit ->
        true
end
```


Recepção selectiva

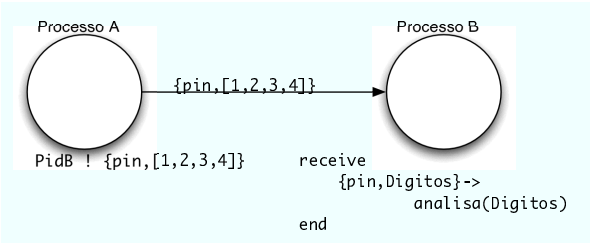
- `Receive` suspende o processo até que uma mensagem correspondente seja recebida
- As mensagens que não correspondem vão sendo guardadas
- Isto pode ser usado para a recepção selectiva de mensagens



- A mensagem `hello` é recebida antes da mensagem bye qualquer que seja a ordem pela qual elas foram enviadas

Envio de dados

- As mensagens podem levar dados:



Identidade

- Um processo recebe os *Pids* dos processos que cria através da função `spawn/3`
- Se quiser que os processos que criou comuniquem com ele deve passar-lhes o seu *Pid*
- `self()` – Devolve o pid do processo que executou esta função.

Exemplo

Um padrão comum

```
start() -> spawn(m, init, [...]).
init(...) -> <initialization>,
             loop(...).
loop(...) -> receive
      stop -> true;
      Pattern1 ->
             <actions>
             loop(...);
             ...
      PatternN ->
             <actions>
             loop(...)
 end.
```


Contador

Servidor com um contador

```
loop(Counter)-> receive
      stop->true;
      {inc,From} ->
             NewCounter=Counter+1,
             loop(NewCounter);
      {dec,From} ->
             NewCounter=Counter-1;
             loop(NewCounter);
      {query,From} ->
             From ! {value,Counter},
             loop(Counter)
      end.
```

echo

```
-module(echo).
-export([start/0]).
-export([pid1/1, pid2/0]).
start() -> Pid2 = spawn(echo, pid2, []),
         spawn(echo, pid1, [Pid2]).
pid1(Pid2) ->
     Pid2 ! {self(), hello},
     receive
     {Pid2, Msg} -> io:format(" P1 got echo from P2~n", []),
             Pid2 ! stop
 end.
pid2() ->
     receive
     {Pid1, Msg} -> Pid1 ! {self(), Msg},
             pid2();
     stop ->  true
 end.
```


Processos registados

- `register(Nome,Pid)`
  - Regista o processo `Pid` com o nome global `Nome`
  - Qualquer processo pode enviar uma mensagem a um processo registado com ≪!≫

```
...
register(ernie,Pid),
...

...
ernie ! Hello
...
```


Timeouts

```
...
receive
        hello -> io:format("hello",[])
        after 1000 -> true
end
% 1000 ms = 1Segundo
....
```

Alarmes

Os timeouts podem ser usados para suspender um processo ou activar alarmes:

```
%%% sleep(T) - processo suspenso durante T ms.

sleep(T) -> receive
                after T ->true
            end.

%%% set_alarm(T, Alarm) - A mensagem Alarm é
%%% enviada ao processo que o activou daí a  T ms.

set_ alarm(T,Alarm) ->
      spawn(timer,alarm,[self(),T,Alarm]).

alarm(Pid, T, Alarm) ->
      receive
              after T -> Pid ! Alarm
      end.
```