Erros em Erlang

Paulo Ferreira
paf@dei.isep.ipp.pt

Fevereiro de 2006

Introdução

- Os programas podem conter erros lógicos que não são detectados pelo compilador
- O Erlang fornece mecanismos de detecção e tratamento de erros em *run-time*
- Primitivas `catch` e `throw`
  - Proteger o código de erros – `catch`
  - Tratamento de excepções – `catch` e `throw`

catch

`catch Expressão`

- Se a avaliação de `Expressão` tiver sucesso devolve o valor de `Expressão`
- Se a avaliação de `Expressão` falhar devolve o tuplo `{'EXIT',Razão}`
  - `Razão` contém a explicação do erro segundo o Erlang
- Desta forma o processo não termina devido ao erro
- Exemplos:
  - `catch 1 + 3` $\Rightarrow$ `4`
  - `catch 1 + a` $\Rightarrow$ `{'EXIT',{badarith,...}}`
  - `catch somar(A,B)` $\Rightarrow$ depende dos valores de `A` e `B`

throw

`throw(Excepção)`

- Retorna uma excepção a um `catch` associado
- Isto funciona para vários níveis abaixo do `catch`
- A excepção propaga-se até ser encontrado um `catch`
- Se não existir um `catch` ocorre um erro de execução
  - Isto pode que o que nós queríamos evitar
  - Ou pode ser o comportamento desejado...

Advertências

- `catch` e `throw` funcionam dentro de um mesmo processo
- Se não se sabe ao certo o que se quer fazer com `catch` e `throw` é melhor não utilizar nenhum dos dois.
- A filosofia do Erlang relativamente a erros é que o programa deve estourar o mais cedo possível
- Desta forma pode-se detectar e corrigir o problema o mais cedo possível
- Logo não se deve fazer em Erlang a clássica ≪detecção de erros≫ típica de outras linguagens de programação.

Exemplo

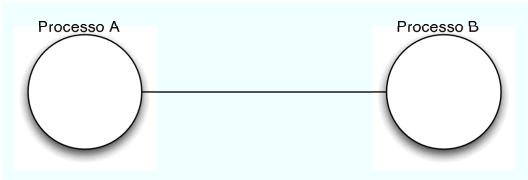
```
test(X) ->
  catch test1(X),
  'processo_nao_morreu'.
test1(X) ->
   ...,
   test2(X).
test2(X) when list(X) ->
   throw({erro,'nao sei processar listas'});
test2(X) ->
   processar(X).
```

- O processo trata as excepções e não termina
- Sem o `catch` é gerado um erro de execução se for lançada a excepção e o processo termina

Links slide 8

Links

Os processos podem ser ligados uns aos outros (um a um):



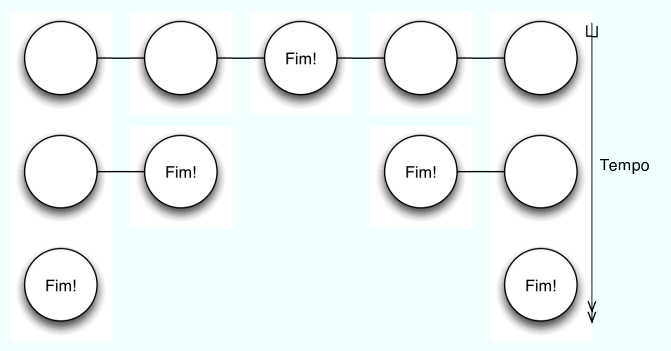
- Os links são criados explícitamente usando:
  - `link(Pid)` ou
  - `spawn_link(Módulo,Função,Argumentos)`
- Os links são bidireccionais
- Podem ser removidos usando
  - `unlink(Pid)`

Sinais de exit

- Quando um processo termina, são enviados sinais de exit a todos os processos aos quais o processo esteja conectado por links.
- Note-se a diferença entre mensagens e sinais
- Quando é que um processo termina?
  - Normalmente (não tem mais código para executar)
  - Devido a um erro de run-time: uma chamada errada a uma função,uma instanciação errada, etc...
  - Quando encontrar uma instrução para terminar

Catástrofe Global

- Quando um processo termina de uma forma anormal, envia sinais de exit a todos os processos com os quais possui links
- Os processos ligados também terminam, passando sinais de exit aos que se encontram ligados, propagando os sinais de exit



Tipos de `exit`

- `exit(Razão)`
  - Termina um processo emitindo um sinal de exit com a razão `Razão`
- `exit(Pid,Razão)`
  - Emite um sinal de exit com a razão `Razão` para o processo `Pid`
  - A função não tem efeitos no processo que a chama
  - Repetindo: `exit/2` não termina a função!
  - Os processos não precisam de estar conectados por links 

Diferentes razões

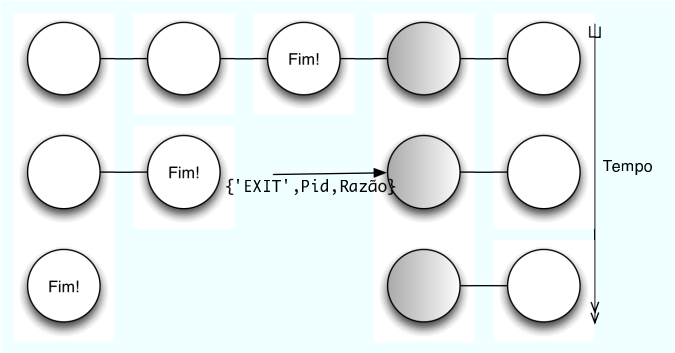
- `normal`
  - Emitida quando um processo termina normalmente
  - Sinais de exit com a razão `normal` não se propagam
- `kill`
  - Tem de ser emitida usando `exit/2`
  - Termina o processo que a recebe incondicionalmente
  - O processo que o recebe, envia sinais com a razão `killed`
- `Outra`
  - Qualquer dos outros tipos é uma razão anormal, e propaga-se

Detecção de erros

- Os processos podem detectar sinais de exit
  - `process_flag(trap_exit,true)`
- Dessa forma, os sinais de exit são transformados em mensagens, com a seguinte forma:
  - `{'EXIT',Pid,Razão}`

Detecção Activada

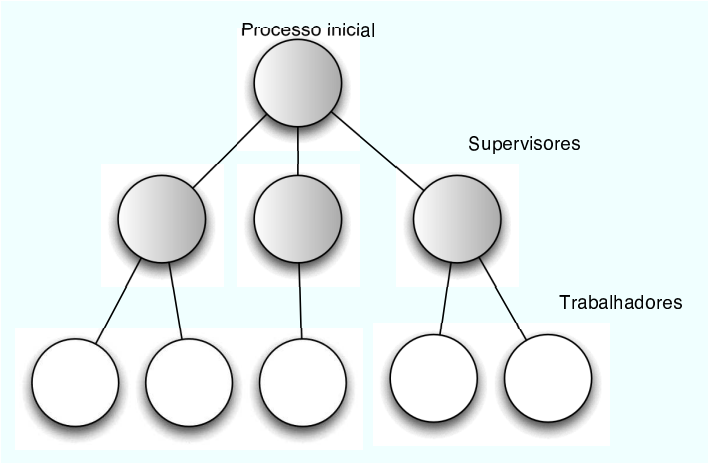
Neste caso supomos que os processos a cinzentos têm a detecção de erros activada:



Resumo

- Processos sem detecção
  - `normal` – Ignora o sinal
  - `kill` – Morre e emite sinal com razão `killed`
  - `Outra` – Morre e emite sinal com razão `Outra`
- Processos com detecção
  - `normal` – Recebe: {'EXIT',Pid,normal}
  - `kill` – Morre e emite sinal com razão `killed`
  - `Outra` – Recebe: {'EXIT',Pid,Razão}

Sistemas robustos



Servidor Robusto 1

```
-module(robust_server).
-export([start/1, init/1]).

start(Resources) ->
        spawn(robust_server, init, [Resources]).

init(Resources) ->
        process_flag(trap_exit, true),
        loop(Resources, []).
```

Servidor Robusto 2

```
loop(Free, Allocated) ->
receive
        {alloc, Pid} ->{Resource, NewFree, NewAllocated} =
                alloc(Pid, Free, Allocated),
                Pid ! {resource, Resource},
                link(Pid),
                loop(NewFree, NewAllocated);
        {free, Resource, Pid} -> {NewFree, NewAllocated} =
                free(Pid, Resource, Free, Allocated),
                unlink(Pid),
                loop(NewFree, NewAllocated);
        {'EXIT', Pid, _Reason} ->{NewFree, NewAllocated} =
                free_all(Pid, Free, Allocated),
                loop(NewFree, NewAllocated)
        end.
```


Servidor Robusto 3

```
alloc(Pid, [Resource| Rest], Allocated) ->
        {Resource, Rest, [{ Pid, Resource}| Allocated]}.

free(Pid, Resource, Free, Allocated) ->
        NewAllocated =
        lists:delete({Pid, Resource}, Allocated),
        {[Resource|Free], NewAllocated}.

free_ all( Pid, Free, Allocated) ->
        free_ all( Pid, Free, [], Allocated).

free_ all( Pid, Free, Allocated,[{ Pid, Resource}| Rest]) ->
        free_ all( Pid, [Resource| Free], Allocated, Rest);
free_ all( Pid, Free, Allocated, [First| Rest]) ->
        free_ all( Pid, Free, [First| Allocated], Rest);
free_ all( Pid, Free, Allocated, []) ->
        {Free, Allocated}.
```

Controlador 1

```
start_server(Jobs) ->
       Pid = spawn(?MODULE,server_init,[Jobs]),
       register(server,Pid).

server_init(Jobs) ->
       process_flag(trap_exit,true),
       Workers = create_workers(Jobs),
       server_loop(Workers).

create_workers([]) ->
       [];

create_workers([Job|T]) ->
       Worker = spawn_link(?MODULE,worker,[Job]),
       [{Worker,Job}|create_workers(T)].
```


Controlador 2

```
server_loop(Workers) ->
       receive
         {'EXIT',Pid,Why} ->
            NewWorkers = recreate_worker(Pid,Workers),
            server_loop(NewWorkers)
       end.

recreate_worker(Pid,[{Pid,Job}|Workers]) ->
       NewWorker = spawn_link(?MODULE,worker,[Job]),
       [{NewWorker,Job}|Workers];

recreate_worker(Pid,[H|Workers]) ->
       [H|recreate_worker(Workers)].

worker(Job) ->
       do_some_work(Job).
```