2023-11-13 - Gestione errori, Link, Monitor, Processi di sistema

Errori in programmi concorrenti

In **sistemi distribuiti**, è necessario implementare dei **sistemi di gestione degli errori**, in modo da **coordinare** la *risposta* o il *crash* di tutti i processi (attori) e non continuare l'esecuzione solo *parziale* o su stato *inconsistente*.

Link e Exit signals

La funzione `link` permette agli attori di **monitorarsi** a vicenda. Quando due attori sono collegati, quando uno dei due **termina**, viene mandato un **segnale** all'altro.

```
% viene creato un collegamento tra il processo corrente e il processo Pid
link(Pid).
```

I link gestiscono, quindi, la **propagazione** di errori tra due attori. Il limite di link tra **due attori è 1**, ma ogni attore può essere collegato ad **infiniti** altri attori. Tutti gli attori *"linkati"* (connessi) ad un attore sono chiamati **link set**.

Quando un attore "muore" *(termina)* **manda un segnale** al suo link set, con il **motivo** della terminazione, nel formato `{'EXIT', Pid, Reason}`:

- messaggio esplicitato dalla terminazione esplicita (`exit(Reason)`)
- messaggio di errore (ad esempio `badarith`, `[{module,fun,2}]}` in caso di divisione per 0)
- `normal` in caso di uscita "naturale"

Un attore può **fingere** la sua terminazione attraverso la sintassi `exit(Pid, Reason)`, che invia il **segnale** `Reason` all'attore specificato da `Pid`, senza terminare.

Processi di sistema

Quando un processo *(normale)* riceve un segnale (diverso da `normal`) **termina a sua volta**.
Un **processo di sistema**, invece, quando riceve un qualsiasi segnale (diverso da `kill`) **NON termina**, ma tratta il segnale come un *normale messaggio* nella sua coda.

Per trasformare un processo normale in un processo di sistema, è necessario eseguire `process_flag(trap_exit, true)`.

Comportamento del **ricevitore** del segnale:

processo di sistema	exit signal	comportamento
true	kill	termina e invia al link set il segnale ricevuto
true	*	aggiunge `{'EXIT', Pid, X}` alla coda dei messaggi
false	normal	continua l'esecuzione e il segnale sparisce
false	*	termina e invia al link set il segnale ricevuto

Esistono delle funzioni per effettuare **link già allo spawn** dell'attore:

- `Pid = spawn(fun() -> ... end)`: i due processi sono indipendenti, non interessa cosa succede al processo creato
- `Pid = spawn_link(fun() -> ... end)`: i due processi sono collegati, se il processo creato termina in errore, termina anche il creatore
- `process_flag(trap_exit, true), Pid = spawn_link(fun() -> ... end)`: i due processi sono collegati ed il creatore è un processo di sistema, se il processo creato termina in errore, il creatore lo gestisce senza terminare

Monitor

I **link sono simmetrici**, ovvero entrambi i processi collegati si controllano **a vicenda**, qualsiasi dei due termini, viene mandato il segnale all'altro. Il numero di link tra due attori è limitato ad 1. Il segnale ricevuto è `{'EXIT', ...}`.

I **monitor** sono link **asimmetrici**, solo un attore controlla un altro attore. Se `A` monitora `B`, in caso `B` termini `A` riceve un segnale, ma se `A` termina, `B` non ne viene a conoscenza.
Il **numero di monitor** tra due attori è **illimitato**, per questo motivo il segnale scatenato da un atomo contiene anche una referenza (`Ref`) al monitor creato: `{'DOWN', Ref, process, B, Reason}`.

```
% l'atomo process è sempre necessario quando si collegano due attori
monitor(process, Pid).
```

Esempio Programma per *"spawnare"* 3 processi con **privilegi** differenti e che effettuano **operazioni** e **terminano** in modo differente.

```
-module(errors).
-export([start/2]).

start(Bool, M) ->
  A = spawn(fun() -> a() end),
  B = spawn(fun() -> b(A, Bool) end),
  C = spawn(fun() -> c(B, M) end),
  sleep(1000), status(a, A), status(b, B), status(c, C).

% always system process
a() ->
  process_flag(trap_exit, true),
  listen(a).

% linked to A, system process depending on Bool
b(A, Bool) ->
  process_flag(trap_exit, Bool),
  link(A),
  listen(b).

% linked to B, normal process, dies in various ways depending on M
c(B, M) ->
  link(B),
  case M of
    {die, Reason} -> exit(Reason);
    {divide, N} -> _ = 1/N, listen(c);
    normal -> true
  end.

% prints the messages received by Prog
listen(Prog) ->
  receive
    Any -> io:format("Process ~p received ~p~n", [Prog, Any]),
    listen(Prog)
  end.

% pauses esecution for T milliseconds
sleep(T) ->
  receive
    after T -> true
  end.

% prints the status (alive or dead) of a process (Pid)
status(Name, Pid) ->
  case erlang:is_process_alive(Pid) of
    true -> io:format("Process ~p (~p) is alive~n", [Name, Pid]);
    false -> io:format("Process ~p (~p) is dead~n", [Name, Pid])
  end.
```

Lanciando `start` con diversi parametri:

- `c` termina in modo **naturale**, quindi `b` non muore:

```
errors:start(false, normal).
% Process a (<0.89.0>) is alive
% Process b (<0.90.0>) is alive
% Process c (<0.91.0>) is dead
```

- `c` termina in modo **non naturale**, quindi `b` riceve il suo segnale, **muore** a sua volta (dato che **non** è un processo di sistema) e manda il segnale ad `a`, che **non muore** (dato che è un processo di sistema):

```
errors:start(false, {die, Mroto}).
% Process a received {'EXIT',<0.95.0>,mroto}
% Process a (<0.94.0>) is alive
% Process b (<0.95.0>) is dead
% Process c (<0.96.0>) is dead
```

- `c` termina in modo **non naturale**, causato da una *divisione per 0*, quindi `b` riceve il suo segnale, **muore** a sua volta (dato che **non** è un processo di sistema) e manda il segnale ad `a`, che **non muore** (dato che è un processo di sistema):

```
errors:start(false, {divide, 0}).
% Process a received {'EXIT',<0.99.0>,
%                       {badarith,
%                           [{errors,c,2,[{file,"errors.erl"},{line,26}]}]}}
% Process a (<0.98.0>) is alive
% Process b (<0.99.0>) is dead
% Process c (<0.100.0>) is dead
```

- `c` termina in modo **non naturale**, quindi `b` riceve il suo segnale, che **non muore** (dato che è un processo di sistema):

```
errors:start(true, {die, mroto}).
% Process b received {'EXIT',<0.114.0>,mroto}
% Process a (<0.112.0>) is alive
% Process b (<0.113.0>) is alive
% Process c (<0.114.0>) is dead
```