LINKS IN ERLANG

A link is a specific kind of relationship that can be created between two processes. When that relationship is set up and one of the processes dies from an unexpected throw, error or exit (see Errors and Exceptions), the other linked process also dies.
This is a useful concept from the perspective of failing as soon as possible to stop errors: if the process that has an error crashes but those that depend on it don't, then all these depending processes now have to deal with a dependency disappearing. Letting them die and then restarting the whole group is usually an acceptable alternative. Links let us do exactly this.

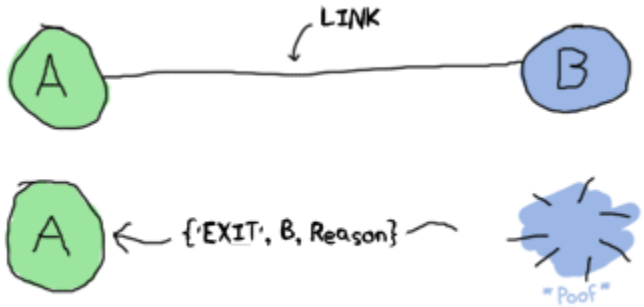
To set a link between two processes, Erlang has the primitive function link/1, which takes a Pid as an argument. When called, the function will create a link between the current process and the one identified by *Pid*. To get rid of a link, use unlink/1. When one of the linked processes crashes, a special kind of message is sent, with information relative to what happened. No such message is sent if the process dies of natural causes (read: is done running its functions.) I'll first introduce this new function as part of linkmon.erl:

```
myproc() ->
timer:sleep(5000),
exit(reason).
```

If you try the next following calls (and wait 5 seconds between each spawn command), you should see the shell crashing for 'reason' only when a link has been set between the two processes.

```
1> c(linkmon).
{ok,linkmon}
2> spawn(fun linkmon:myproc/0).
<0.52.0>
3> link(spawn(fun linkmon:myproc/0)).
true
** exception error: reason
```

Or, to put it in picture:



However, this `{'EXIT', B, Reason}` message can not be caught with a `try ... catch` as usual. Other mechanisms need to be used to do this. We'll see them later.

It's important to note that links are used to establish larger groups of processes that should all die together:

```
chain(0) ->
receive
_ -> ok
after 2000 ->
exit("chain dies here")
end;
chain(N) ->
Pid = spawn(fun() -> chain(N-1) end),
link(Pid),
receive
_ -> ok
end.
```

This function will take an integer *N*, start *N* processes linked one to the other. In order to be able to pass the *N-1*argument to the next 'chain' process (which calls `spawn/1`), I wrap the call inside an anonymous function so it doesn't need arguments anymore. Calling `spawn(?MODULE, chain, [N-1])` would have done a similar job.

Here, I'll have many processes linked together, dying as each of their successors exits:

```
4> c(linkmon).
{ok,linkmon}
5> link(spawn(linkmon, chain, [3])).
true
** exception error: "chain dies here"
```

And as you can see, the shell does receive the death signal from some other process. Here's a drawn representation of the spawned processes and links going down:

```
[shell] == [3] == [2] == [1] == [0]

[shell] == [3] == [2] == [1] == *dead*

[shell] == [3] == [2] == *dead*

[shell] == [3] == *dead*

[shell] == *dead*
```

```
*dead, error message shown*

[shell] <-- restarted
```

After the process running `linkmon:chain(0)` dies, the error is propagated down the chain of links until the shell process itself dies because of it. The crash could have happened in any of the linked processes; because links are bidirectional, you only need one of them to die for the others to follow suit.

Note: If you wanted to kill another process from the shell, you could use the function exit/2, which is called this way: `exit(Pid, Reason)`. Try it if you wish.

Note: Links can not be stacked. If you call `link/1` 15 times for the same two processes, only one link will still exist between them and a single call to `unlink/1` will be enough to tear it down.

Its important to note that `link(spawn(Function))` or `link(spawn(M,F,A))` happens in more than one step. In some cases, it is possible for a process to die before the link has been set up and then provoke unexpected behavior. For this reason, the function spawn_link/1-3 has been added to the language. It takes the same arguments as`spawn/1-3`, creates a process and links it as if `link/1` had been there, except it's all done as an atomic operation (the operations are combined as a single one, which can either fail or succeed, but nothing else). This is generally considered safer and you save a set of parentheses too.

Source : http://learnyousomeerlang.com/errors-and-processes