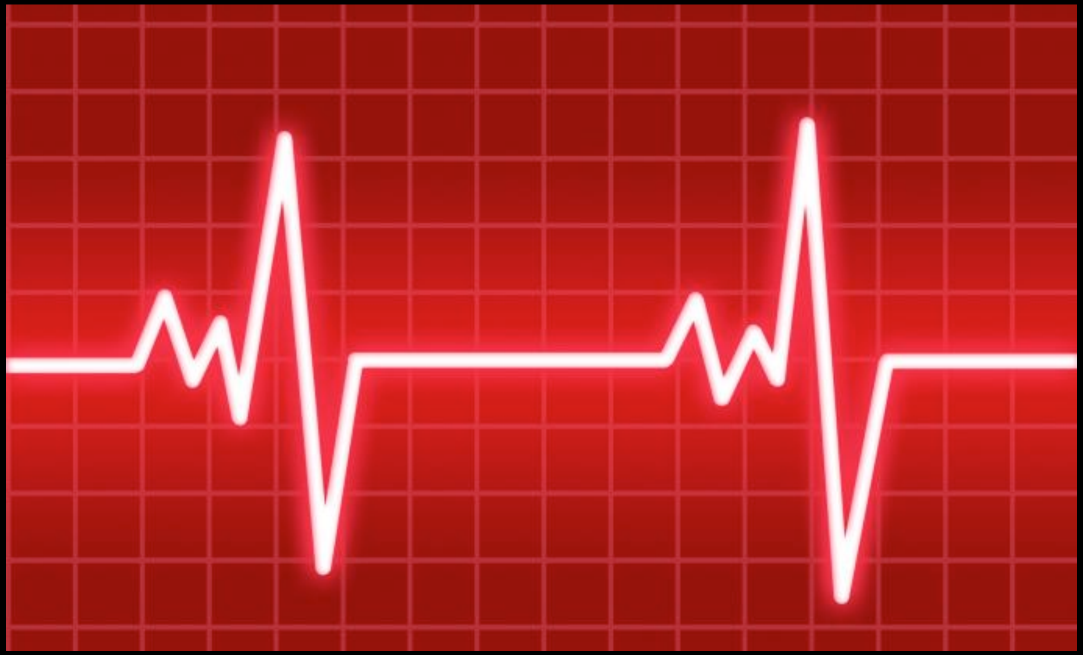
Process supervision in Erlang

Module 8 - Erlang tutorial 4



Operating systems and process oriented programming 2020

1DT096

Spring 2020

karl.marklund@it.uu.se

Uppsala university

Process termination

Process termination

When a process terminates, it always terminates with an **exit reason**.

- ★ A process is said to terminate normally, if the exit reason is the **atom normal**.
- ★ A process with no more code to execute terminates normally.

`exit(Reason) -> no_return()`

Types

```
Reason = term()
```

Description

Stops the execution of the calling process with exit reason **Reason**, where **Reason** is any term.

Return value

Since evaluating this function causes the process to terminate, it has no return value.

Example: Terminate the Erlang shell.

```
1> exit(done).
** exception exit: done
2>
```

★ The shell is terminated with the **atom** **done** as reason.

★ The shell is automatically re-started.

```
2> exit({done, 127}).
** exception exit: {done,127}
3>
```

★ The shell is terminated with the **tuple** **{done, 127}** as reason.

★ The shell is automatically re-started.

`exit(Pid, Reason) -> true`

Types

```
Pid = pid()
Reason = term()
```

Description

Sends an exit signal with exit reason Reason to the process or port identified by Pid.

Return value

Always returns true.

Example: Terminate the Erlang shell.

```
1> exit(self(), done).
** exception exit: done
2>
```

★ The PID of the shell is obtained by calling self().

★ The shell is terminated with the **atom** **done** as reason.

★ The shell is automatically re-started.

```
2> exit(self(), {done, 127}).
** exception exit: {done,127}
3>
```

★ The shell is terminated with the **tuple** **{done, 127}** as reason.

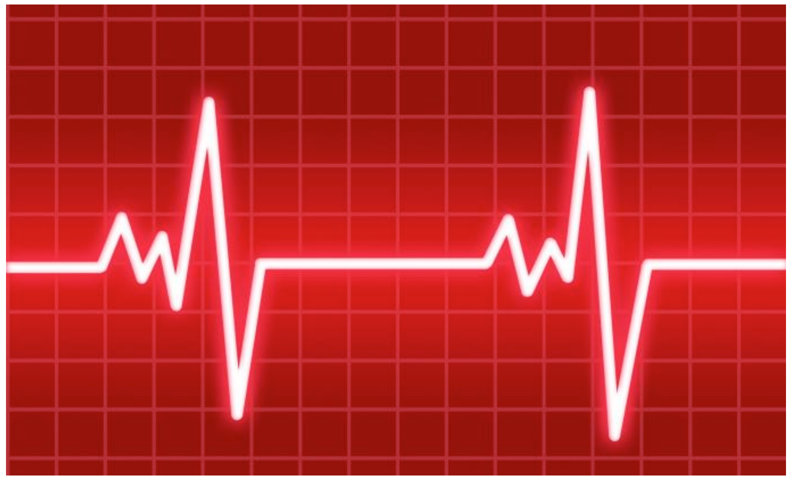
★ The shell is automatically re-started.

Process supervision

Fault tolerance

What can be done if a process dies unexpectedly?

Erlang comes with built in **process supervision.**



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 **linked processes dies** unexpectedly, the **other** linked process **also dies**.

This is a useful concept from the perspective of failing as soon as possible, aka **crash and burn**, to stop errors.

`link(Pid) -> true`

Types

```
Pid = pid()
```

Description

Creates a link between the calling process and another process, if there is not such a link already. If a process attempts to create a link to itself, nothing is done.

Return value

Always returns true.

`spawn_link(Fun) -> pid()`

Types

`Fun = function()`

Description

Returns the process identifier of a new process started by the application of `Fun` to the empty list `[]`. A link is created between the calling process and the new process, atomically.Return value

`unlink(Pid) -> true`

Types

```
Pid = pid()
```

Description

Removes the link, if there is one, between the calling process and the process referred to by Pid.

Return value

Always returns true.

Free all bound variables and re-start the pingpong process.

```
1> f(), Pid = pingpong:start().
<0.67.0>
```

Create a link between the shell and the pingpong process.

```
2> link(Pid).
true
```

Send an unsupported message to the pingpong process.

```
3> Pid ! hello.
<0.67.0> received unsupported message hello
hello
** exception exit: unsupported_message
```

After creating a link between the shell and the pingpong process, when the pingpong process **terminates abnormally** with **reason** **unsupported_message**, the shell terminates with the same reason.

Register a process with a name

`register(RegName, Pid) -> true`

Types

`RegName = atom()`

`Pid = pid()`

Description

Associates the name **RegName** with a process identifier (**pid**).

RegName, which must be an atom, can be used instead of the pid or port identifier in send operator.

```
RegName ! Message.
```

Return value

On success, always returns true.

`unregister(RegName) -> true`

Types

```
RegName = atom()
```

Description

Removes the registered name RegName associated with a pid.

Return value

On success, always returns true.

`whereis(RegName) -> pid() | undefined`

Types

```
RegName = atom()
```

Description

Returns the **pid** of the process registered under the **name RegName**.

Returns **undefined** if the name is not registered.

Return value

Returns true on success.

A bomb process

Create a bomb

An experiment where we use the BIF **`exit(Reason)`** to terminate unexpectedly.

```
-module(bomb).
-export([start/1, test/1, test/2, chain/2]).

start(0) ->
    io:format("0 >> Booom <<~n"),
    exit(boom);
start(Seconds) ->
    timer:sleep(1000),
    io:format("~w...", [Seconds]),
    start(Seconds - 1).

test(Seconds, link) ->
    link(spawn(?MODULE, start, [Seconds])).
test(Seconds) ->
    spawn(?MODULE, start, [Seconds]).
```

Create a bomb

An experiment where we use the BIF **`exit(Reason)`** to terminate unexpectedly.

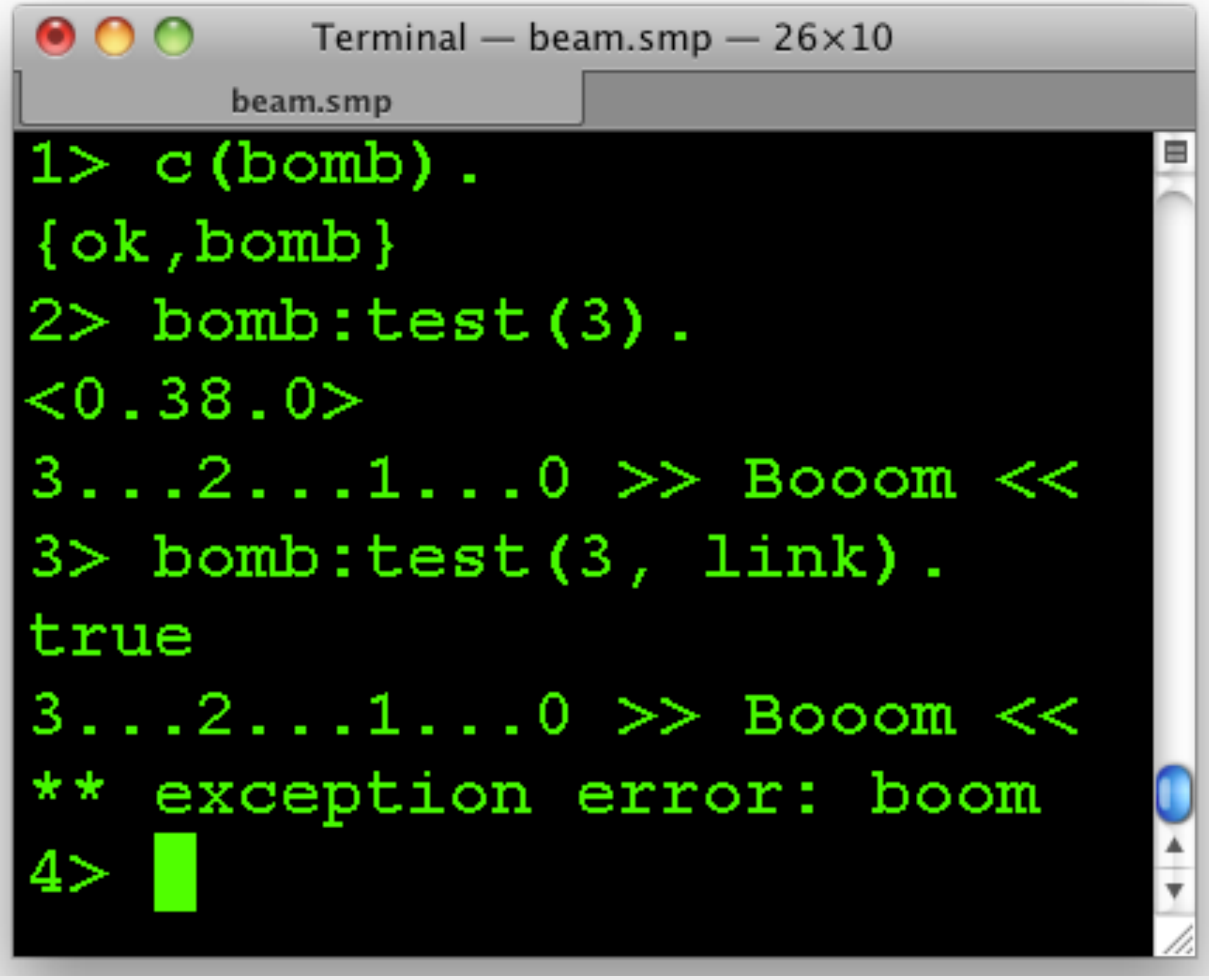
```
-module(bomb).
-export([start/1, test/1, test/2, chain/2]).

start(0) ->
    io:format("0 >> Booom <<~n"),
    exit(boom);
start(Seconds) ->
    timer:sleep(1000),
    io:format("~w...", [Seconds]),
    start(Seconds - 1).

test(Seconds, link) ->
    link(spawn(?MODULE, start, [Seconds])).
test(Seconds) ->
    spawn(?MODULE, start, [Seconds]).
```

`exit(Reason) -> no_return()`

Stops the execution of the calling process with the exit reason Reason, where Reason is any term. Since evaluating this function causes the process to terminate, it has no return value.



```
1> c(bomb).
{ok,bomb}
2> bomb:test(3).
<0.38.0>
3...2...1...0 >> Booom <<
3> bomb:test(3, link).
true
3...2...1...0 >> Booom <<
** exception error: boom
4> 
```

The shell crashes with exception **boom** only when a link has been set between the shell and the bomb processes.

A chain of linked processes

A chain of processes

Let’s create a chain of processes with a bomb at the end of the cain.

```
chain(0, _) ->
    receive
        _ -> ok
    after 2000 ->
            bomb:start(3)
    end;
chain(N, Link) ->
    Pid = spawn(fun() -> chain(N-1, Link) end),
    case Link of
        true -> link(Pid);
        _    -> ok
    end,
    receive
        _ -> ok
    after 10000 ->
            io:format("~p termiantes normally~n", [self()])
    end.
```

Use a **time out** in the receive construct to start the bomb after 2 seconds.

Use a **time out** in the receive construct to terminate normally after 10 seconds.

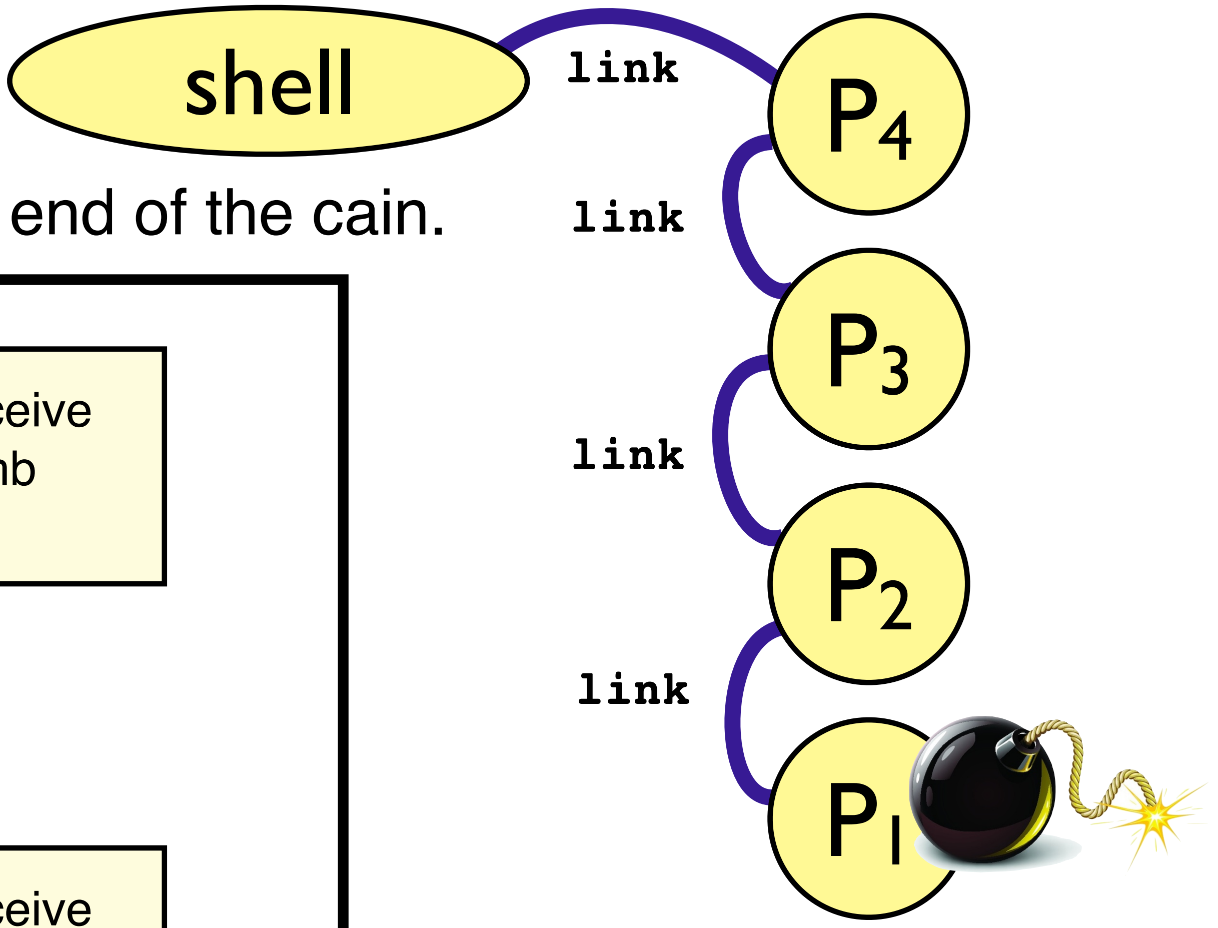
A chain of processes

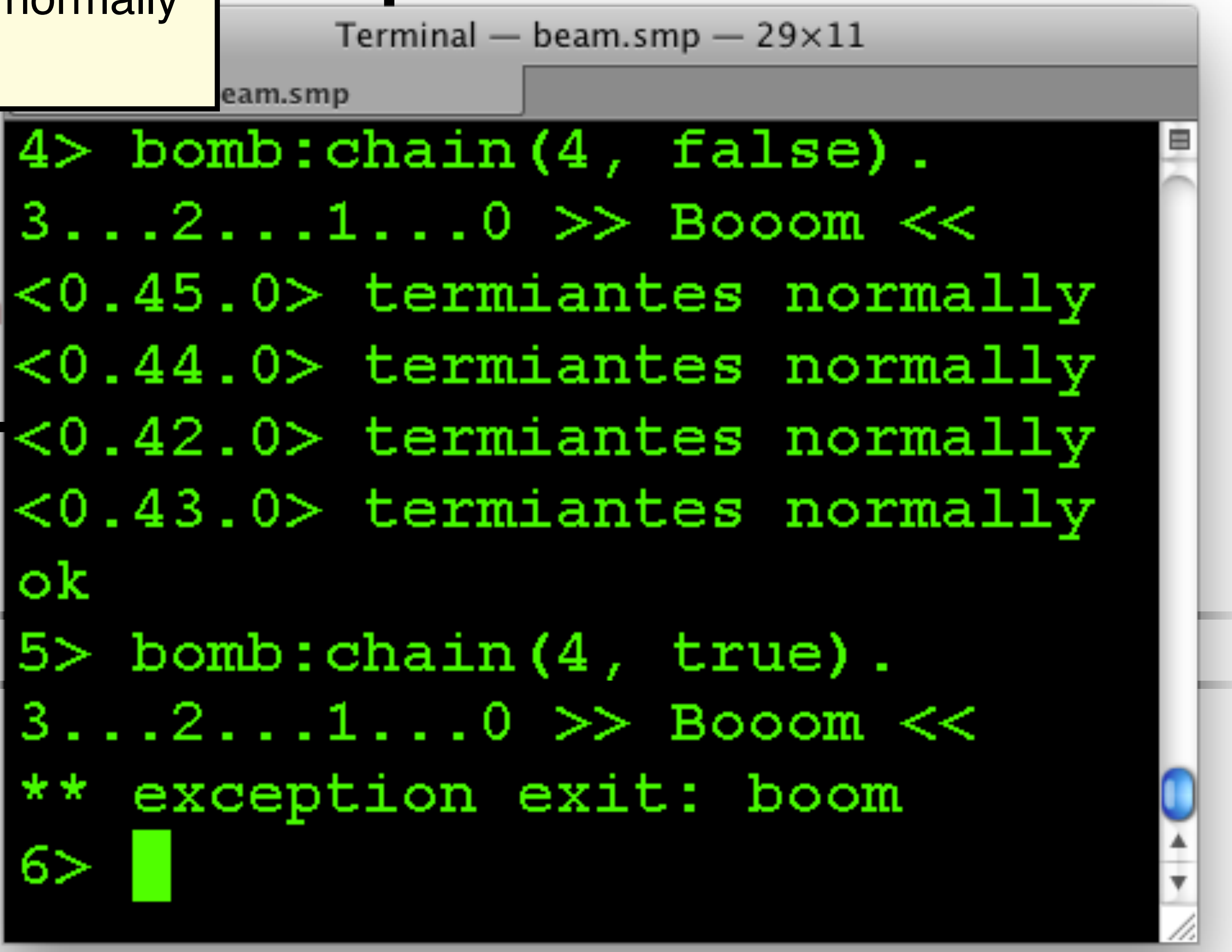
Let's create a chain of processes with a bomb at the end of the cain.

```
chain(0, _) ->
    receive
        _ -> ok
    after 2000 ->
            bomb:start(3)
    end;
chain(N, Link) ->
    Pid = spawn(fun() -> chain(N-1, Link) end),
    case Link of
        true -> link(Pid);
        _    -> ok
    end,
    receive
        _ -> ok
    after 10000 ->
            io:format("~p termiantes normally~n
    end.
```

Use a **time out** in the receive construct to start the bomb after 2 seconds.

Use a **time out** in the receive construct to terminate normally after 10 seconds.





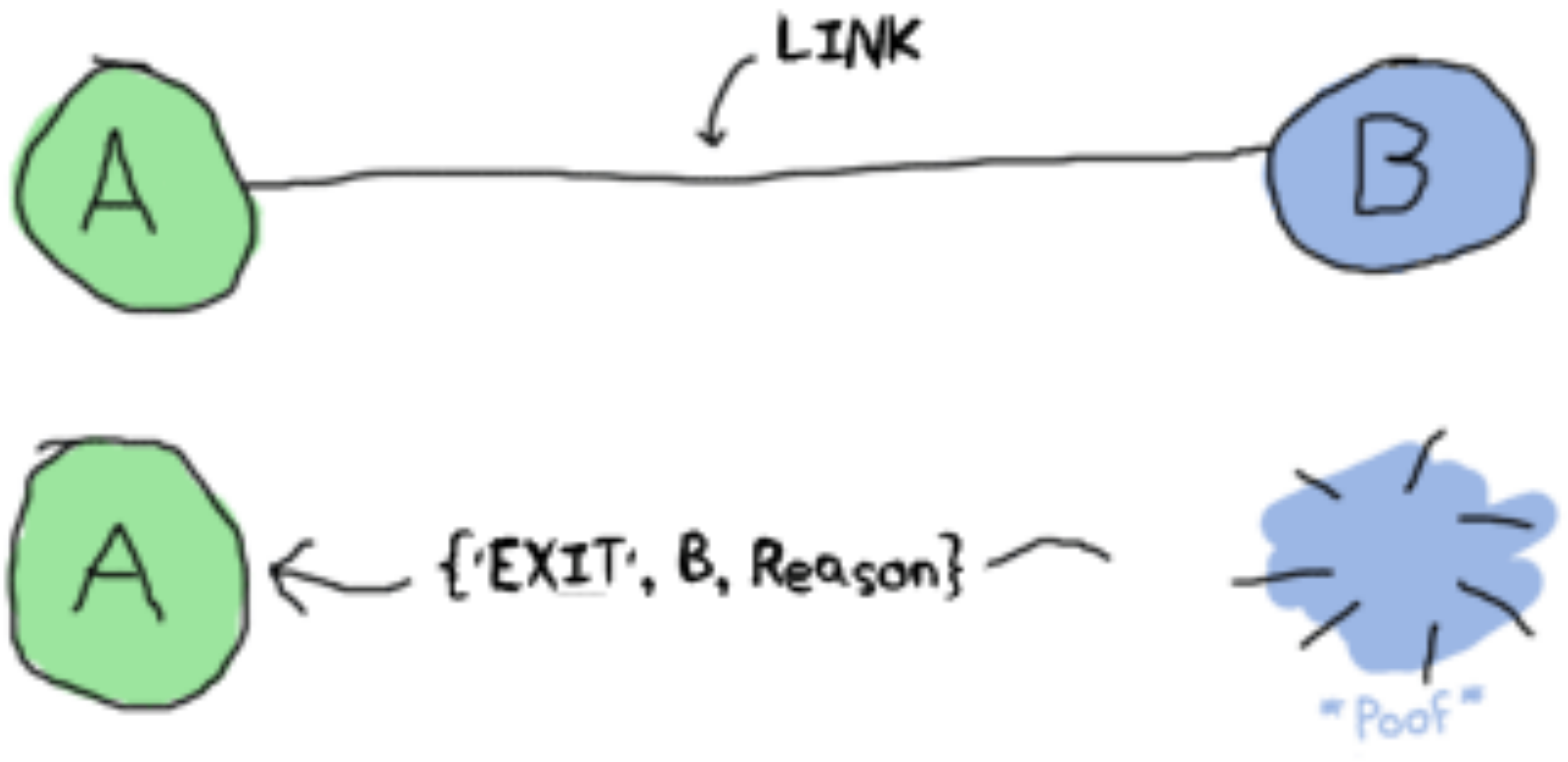
```
4> bomb:chain(4, false).
3...2...1...0 >> Booom <<
<0.45.0> termiantes normally
<0.44.0> termiantes normally
<0.42.0> termiantes normally
<0.43.0> termiantes normally
ok
5> bomb:chain(4, true).
3...2...1...0 >> Booom <<
** exception exit: boom
6> 
```

The shell crashes with exception **boom** only when we created a chain of links

Error propagation

Error propagation across processes is done through a process similar to message passing, but with a special type of message called **signals**.

Exit signals are secret messages that automatically act on processes, killing them in the action.



Source: http://learnyousomeerlang.com/errors-and-processes 2012-03-22

`process_flag(trap_exit, Boolean) -> OldBoolean`

Types

```
Boolean = OldBoolean = boolean()
```

Description

When **trap_exit** is set to **true**, exit signals arriving to a process are converted to **{'EXIT', From, Reason} messages**, which can be received as ordinary messages.

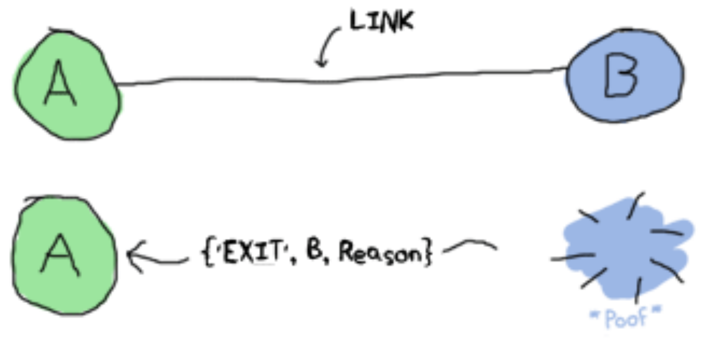
If **trap_exit** is set to **false**, the process exits if it receives an exit signal other than normal and the exit signal is propagated to its linked processes.

Return value

Returns the old value of the flag.

Trap exit

In order to be reliable, an application needs to be able to both kill and restart a process quickly. Right now, links are alright to do the killing part. What's missing is the restarting.



```
test(Seconds, link, trap_exit) ->
    process_flag(trap_exit, true),
    spawn_link(?MODULE, start, [Seconds]),
    receive
        Msg -> Msg
    end.
```

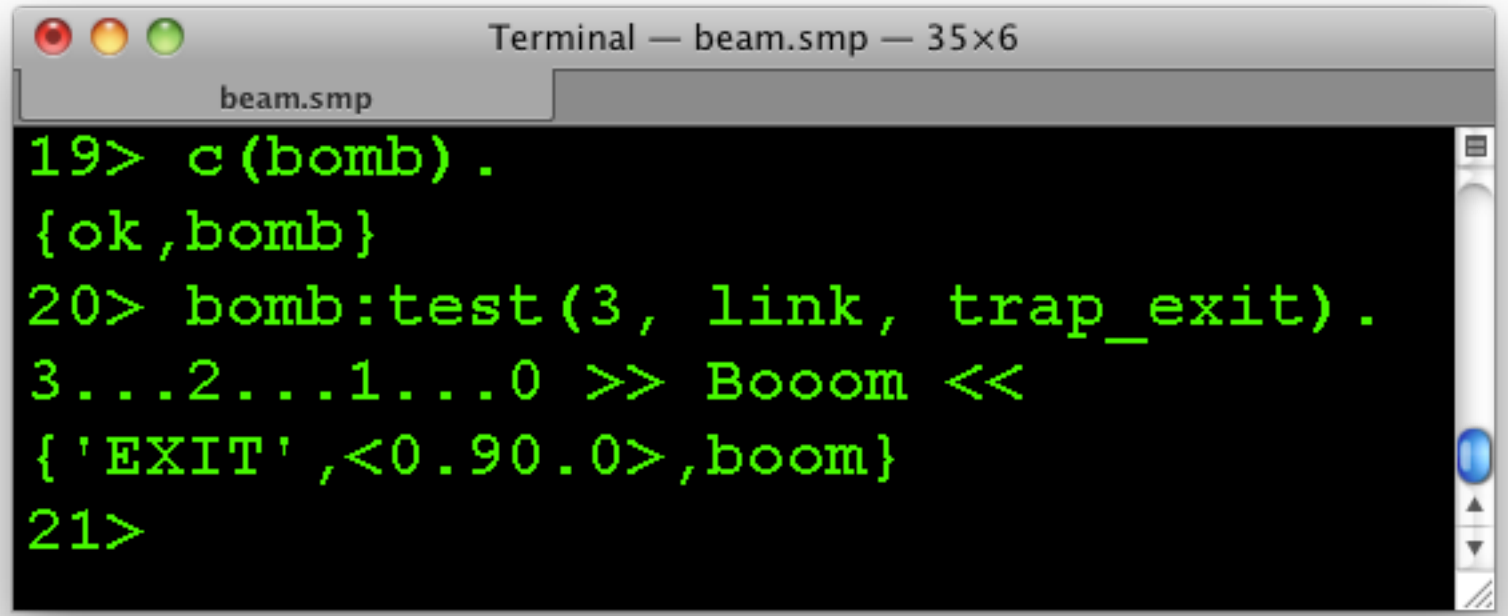
System processes are basically normal processes, except they can convert exit signals to regular messages. This is done by calling:

`process_flag(trap_exit, true)`

, in a running process.

Spawn and set up a link using the BIF `spawn_link/3`.

When the bomb goes off, we now receive a message instead of getting killed.



```
19> c(bomb).
{ok,bomb}
20> bomb:test(3, link, trap_exit).
3...2...1...0 >> Booom <<
{'EXIT',<0.90.0>,boom}
21>
```