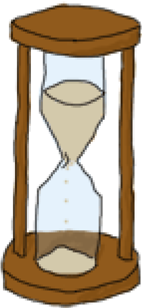
TIME OUT PROGRAM IN ERLANG

Let's try a little something with the help of the command `pid(A,B,C)`, which lets us change the 3 integers *A*, *B* and *C* into a pid. Here we'll deliberately feed `kitchen:take/2` a fake one:

```
20> kitchen:take(pid(0,250,0), dog).
```

Woops. The shell is frozen. This happened because of how `take/2` was implemented. To understand what goes on, let's first revise what happens in the normal case:

1. A message to take food is sent from you (the shell) to the fridge process;
2. Your process switches to receive mode and waits for a new message;
3. The fridge removes the item and sends it to your process;
4. Your process receives it and moves on with its life.



And here's what happens when the shell freezes:

1. A message to take food is sent from you (the shell) to an unknown process;
2. Your process switches to receive mode and waits for a new message;
3. The unknown process either doesn't exist or doesn't expect such a message and does nothing with it;
4. Your shell process is stuck in receive mode.

That's annoying, especially because there is no error handling possible here. Nothing illegal happened, the program is just waiting. In general, anything dealing with asynchronous operations (which is how message passing is done in Erlang) needs a way to give up after a certain period of time if it gets no sign of receiving data. A web browser does it when a page or image takes too long to load, you do it when someone takes too long before answering the phone or is late at a meeting. Erlang certainly has an appropriate mechanism for that, and it's part of the `receive` construct:

```
receive
Match -> Expression1
after Delay ->
Expression2
end.
```

The part in between `receive` and `after` is exactly the same that we already know. The `after` part will be triggered if as much time as *Delay* (an integer representing milliseconds) has been spent without receiving a message that matches the *Match* pattern. When this happens, *Expression2* is executed.

We'll write two new interface functions, `store2/2` and `take2/2`, which will act exactly like `store/2` and `take/2` with the exception that they will stop waiting after 3 seconds:

```
store2(Pid, Food) ->
Pid ! {self(), {store, Food}},
receive
{Pid, Msg} -> Msg
after 3000 ->
timeout
end.

take2(Pid, Food) ->
Pid ! {self(), {take, Food}},
receive
{Pid, Msg} -> Msg
after 3000 ->
timeout
end.
```

Now you can unfreeze the shell with `^G` and try the new interface functions:

```
User switch command
--> k
--> s
--> c
Eshell V5.7.5 (abort with ^G)
1> c(kitchen).
{ok,kitchen}
2> kitchen:take2(pid(0,250,0), dog).
timeout
```

And now it works.

Note: I said that `after` only takes milliseconds as a value, but it is actually possible to use the atom `infinity`. While this is not useful in many cases (you might just remove the `after` clause altogether), it is sometimes used when the programmer can submit the wait time to a function where receiving a result is expected. That way, if the programmer really wants to wait forever, he can.

There are uses to such timers other than giving up after too long. One very simple example is how the`timer:sleep/1` function we've used before works. Here's how it is implemented (let's put it in a new multiproc.erlmodule):

```
sleep(T) ->
receive
after T -> ok
end.
```

In this specific case, no message will ever be matched in the `receive` part of the construct because there is no pattern. Instead, the `after` part of the construct will be called once the delay T has passed.

Another special case is when the timeout is at 0:

```
flush() ->
receive
_ -> flush()
after 0 ->
ok
end.
```

When that happens, the Erlang VM will try and find a message that fits one of the available patterns. In the case above, anything matches. As long as there are messages, the `flush/0` function will recursively call itself until the mailbox is empty. Once this is done, the `after 0 -> ok` part of the code is executed and the function returns.

Source : http://learnyousomeerlang.com/more-on-multiprocessing