Concurrent Programming in Erlang

We have introduced process linking, exit signals, and “trapping exits”, as well as seeing how these can be applied to the frequency server example. In this exercise we’ll look in more detail at the example, and use the ‘observer’ tool to help us see what is going in when parts of a system fail.

Use the comments here to share your approaches and solutions to this exercise. In the next step, we’ll have a discussion about how you’ve got on.

There is a supporting file, `frequency_hardened.erl`, available (as a zip file) under ‘Downloads’ below. This is an update of the `frequency.erl` files from previous steps - the file itself is called `frequency_hardened.erl` to distinguish it from the original, but it still defines the `frequency` module so you may want to re-name the file.

Modelling clients

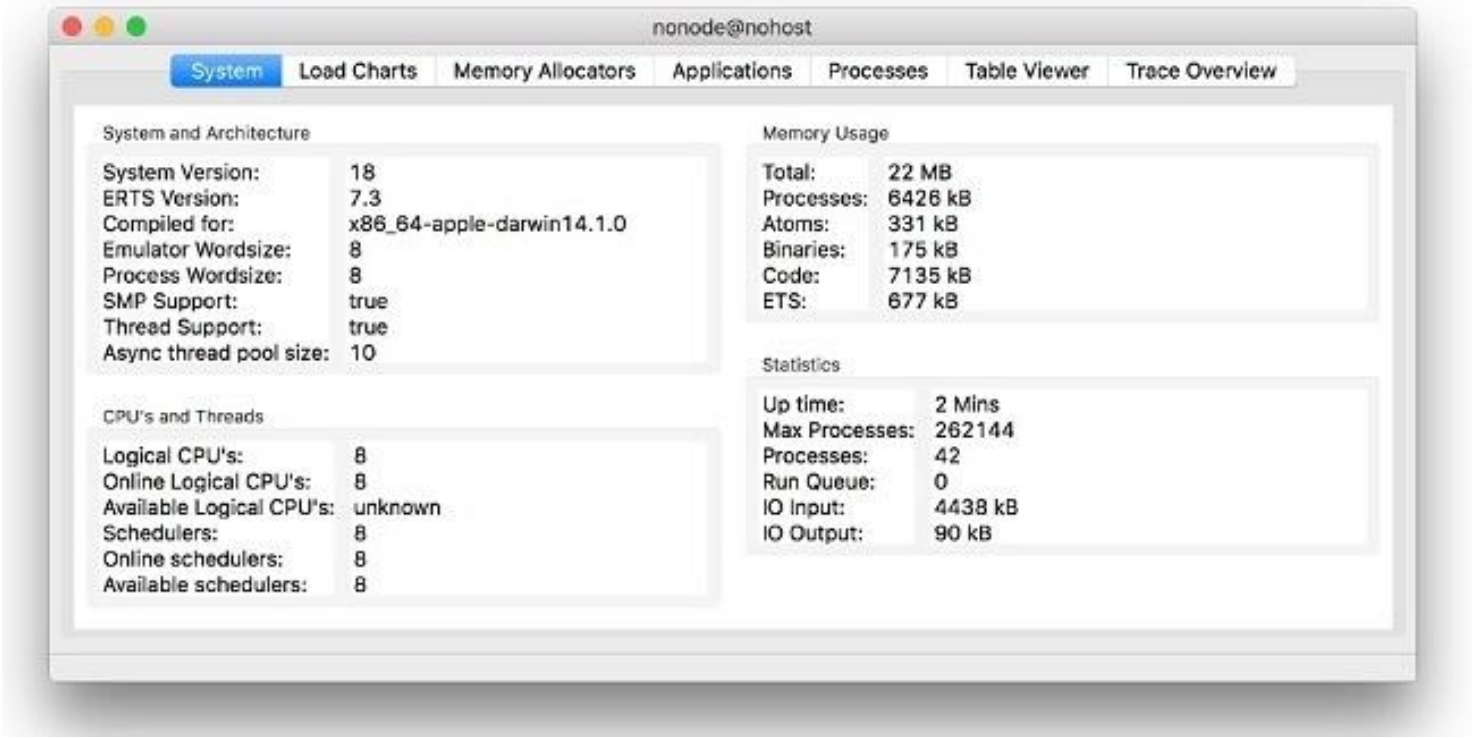
In practice, a server will have multiple clients, and in order to see how the system behaves, implement a `client` function that, when spawned as a process, can be used to model a client. To allow a simulation to go on indefinitely, your function should cycle through a number of allocations and deallocations, forever (since the server is designed to live indefinitely, too).

We would like to model systems where there are multiple clients, so it would be useful to **parameterise** the client so that it can behave in different ways when called with different parameters.

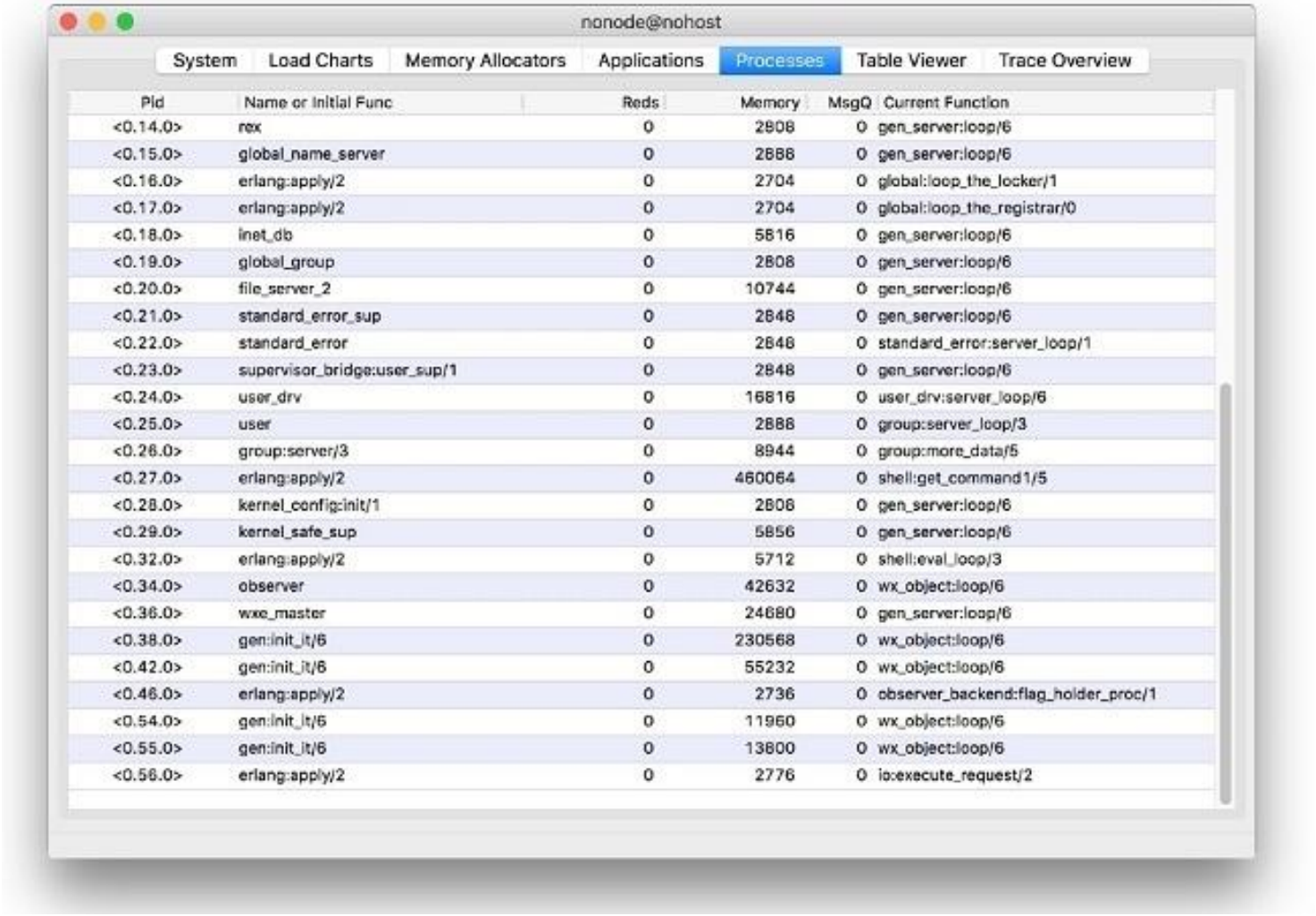
The observer tool

Erlang comes with a GUI-based tool called the observer, which allows us to see, among other things, all the processes running in a system at any time, including their particular `Pid`.

To run the observer, type `observer:start()` in the erlang shell. You will see this tabbed window, which opens with an overview of the system:



Selecting the **Processes** tab will give a view like this:



Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.14.0>	rex	0	2808	0	gen_server:loop/6
<0.15.0>	global_name_server	0	2888	0	gen_server:loop/6
<0.16.0>	erlang:apply/2	0	2704	0	global:loop_the_locker/1
<0.17.0>	erlang:apply/2	0	2704	0	global:loop_the_registrar/0
<0.18.0>	inet_db	0	5816	0	gen_server:loop/6
<0.19.0>	global_group	0	2808	0	gen_server:loop/6
<0.20.0>	file_server_2	0	10744	0	gen_server:loop/6
<0.21.0>	standard_error_sup	0	2848	0	gen_server:loop/6
<0.22.0>	standard_error	0	2848	0	standard_error:server_loop/1
<0.23.0>	supervisor_bridge:user_sup/1	0	2848	0	gen_server:loop/6
<0.24.0>	user_drv	0	16816	0	user_drv:server_loop/6
<0.25.0>	user	0	2888	0	group:server_loop/3
<0.26.0>	group:server/3	0	8944	0	group:more_data/5
<0.27.0>	erlang:apply/2	0	460064	0	shell:get_command1/5
<0.28.0>	kernel_config:init/1	0	2808	0	gen_server:loop/6
<0.29.0>	kernel_safe_sup	0	5856	0	gen_server:loop/6
<0.32.0>	erlang:apply/2	0	5712	0	shell:eval_loop/3
<0.34.0>	observer	0	42632	0	wx_object:loop/6
<0.36.0>	wxe_master	0	24680	0	gen_server:loop/6
<0.38.0>	gen:init_it/6	0	230568	0	wx_object:loop/6
<0.42.0>	gen:init_it/6	0	55232	0	wx_object:loop/6
<0.46.0>	erlang:apply/2	0	2736	0	observer_backend:flag_holder_proc/1
<0.54.0>	gen:init_it/6	0	11960	0	wx_object:loop/6
<0.55.0>	gen:init_it/6	0	13800	0	wx_object:loop/6
<0.56.0>	erlang:apply/2	0	2776	0	io:execute_request/2

in which you can see the processes listed.

Observing the frequency server and clients.

Using the observer, set up a system of frequency server and at least two clients, and kill the frequency server when the clients are in possession of a frequency – you can make the clients “sleep” at any point using the `timer:sleep/1` function – observe that the clients are killed too.

How would you modify the clients so that they are **not** affected by the server terminating? How could you then shut down the entire system if you needed to?
