Starting the System & Basic Erlang Exercises

These exercises will help you get accustomed with the Erlang development and run time environments. Once you have set up the Erlang mode for **emacs**, you will be ready to write your first program.

To start the Erlang shell, type erl when working on Unix environments or double click on the Erlang Icon in Windows Environments. Once your shell has started, you will get some system printouts followed by a prompt.

If you are working in Unix, you should get some thing like this (Possibly with more system printouts).

```
cesarini@ramone> erl
Erlang R14B (erts-5.8.1) [source] [64-bit] [smp:4:4]
 [rq:4] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.8.1  (abort with ^G)
1>
```

Exercise 1: The shell

Type in the following Erlang expressions in the shell. They will show some of the principles (including pattern matching and single variable assignment) described in the lectures. What happens when they execute, what values do the expressions return, and why?

A. Erlang expressions

```
1 + 1.
[1|[2|[3|[]]]].
```

B. Assigning through pattern matching

```
A = 1.
B = 2.
A + B.
A = A + 1.
```

C. Recursive lists definitions

```
L = [A|[2,3]].
[[3,2]|1].
[H|T] = L.
```

D. Flow of execution through pattern matching

```
B = = 2.
B = 2.
2 = B.
B = C.
C = B.
B = C.
```
(repeat it now that C is bound).

E. Extracting values in composite data types through pattern matching

```
Person = {person, "Mike", "Williams", [1,2,3,4]}.
{person, Name, Surname, Phone} = Person.
Name.
```

Exercise 2: Setting up Emacs

This exercise will enable you to use the Erlang mode for emacs, and start it automatically every time you load a file with a .erl or .hrl (Erlang include files) extension. Copy (or modify if it already exists) a file called .emacs from your account directory. You should place it (or find it) in your user root directory if you are working with Unix or in the C:\ directory if you are working with windows. The modifications described in the slides contain all the information you need. The variables you must change include the **load-path**, the **erlang-root-dir** and the **exec-path**. Other variables are optional.

Do not limit yourself to the bare essential. Try to turn on the syntax highlighting automatically and configure your email address.

Hint: If you have not used emacs before, find the emacs tutorial in the menus, and take a brief look at it. You should do it after the course, as it will allow you to make the most out of your editor.

Exercise 3: Modules and Functions

Copy the demo module from the *Modules* example slide in the Basic Erlang course material. Compile it and try and run it from the shell. What happens when you call **demo:times(3,5)**? What about **double(6)** when omitting the module name?

Example:

```
Eshell V5.8.1  (abort with ^G)
1> c(demo).
{ok, demo}
2> demo:double(6).
12
```

Exercise 4: Simple Pattern Matching

Boolean operators compare values that are either true or false and also return true or false, depending on the input. As an example, the 'and' operator will return true if, and only if both operands are 'true'; otherwise, 'false' will be returned. Another operator is 'or', which will return 'true' if any of the operands is 'true':

P	**Q**	P **and** Q	P **or** Q
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

A third operator, called 'not', will simply reverse the input it has received: 'not true' will return 'false' and 'not false' will return 'true'.

Write a module boolean.erl that takes logical expressions and boolean values (represented as the atoms true and false) and returns their boolean result. The functions you should write should include **b_not/1, b_and/2** and **b_or/2**You may not use the logical constructs **and**, **or** or **not**. Test your module from the shell.

Example:

```
b_not(false) ⇒ true
b_and(false, true) ⇒ false
b_and(b_not(b_and(true, false)), true) ⇒ true
```

Note 1: `foo(X) ⇒ Y` means that calling the function foo with the parameter X will result in the value Y being returned.

Note 2: and, or and not are reserved words in Erlang.

Sequential Programming Exercises

These exercises will get you familiar with recursion and its different uses. Pay special attention to the different recursive patters that we covered during the lectures. If you are having problems finding bugs or following the recursion, try using the debugger.

Exercise 1: Evaluating Expressions

A. Write a function sum/1 which given a positive integer N will return the sum of all the integers between 1 and N.
Example: `sum(5) ⇒ 15.`

B. Write a function sum_interval/2 which given two integers N and M, where N =< M, will return the sum of the interval between N and M. If N > M, you want your process to terminate abnormally.
Example: `sum_interval(1,3) ⇒ 6.`
`sum_interval(6,6) ⇒ 6.`

Exercise 2: Creating Lists

A. Write a function which returns a list of the format `[1,2,..,N-1,N].`
Example: `create(3) ⇒ [1,2,3].`

B. Write a function which returns a list of the format `[N, N-1,..,2,1].`
Example: `reverse_create(3) ⇒ [3,2,1].`

Exercise 3: Side effects

A. Write a function which prints out the integers between 1 and N.
Example:

```
print(5).
1
2
3
4
5
ok
```

Hint: Use `io:format("Number:~p~n",[N]).`

B. Write a function which prints out the even integers between 1 and N.
Example:

```
even_print(5).
2
4
ok
```

Hint: Use guards

Exercise 4: Database Handling Using Lists

Write a module **db.erl** that creates a database and is able to store, retrieve and delete elements in it. The function **destroy/1** will delete the database. Considering that Erlang has garbage collection, you do not need to do anything. Had the db module however stored everything on file, you would delete the file. We are including the destroy function so as to make the interface consistent. You may not use the **lists** library module, and have to implement all the recursive functions yourself.

Hint: Use lists and tuples your main data structures. When testing your program, remember that Erlang variables are single assignment.

Interface:

```
db:new() ⇒ DbRef.
db:destroy(DbRef) ⇒ ok.
db:write(Key, Element, DbRef) ⇒ NewDbRef.
db:delete(Key, DbRef) ⇒ NewDbRef.
db:read(Key, DbRef) ⇒{ok, Element} | {error, instance}.
db:match(Element, DbRef) ⇒ [Key1, ..., KeyN].
```

Example:

```
1> c(db).
{ok,db}
2> Db = db:new().
[]
3> Db1 = db:write(francesco, london, Db).
[{francesco,london}]
4> Db2 = db:write(lelle, stockholm, Db1).
[{francesco,london},{lelle,stockholm}]
5> db:read(francesco, Db2).
{ok,london}
6> Db3 = db:write(joern, stockholm, Db2).
[{francesco,london},{lelle,stockholm},{joern,stockholm}]
7> db:read(ola, Db3).
{error,instance}
8> db:match(stockholm, Db3).
[lelle,joern]
9> Db4 = db:delete(lelle, Db3).
[{francesco,london},{joern,stockholm}]
10> db:match(stockholm, Db4).
[joern]
11> db:write(joern, london, Db4).
[{francesco,london},{joern,london}]
```

Note: Due to single assignment of variables in Erlang, we need to assign the updated database to a new variable every time.

Advanced Exercise 5: Manipulating Lists

A. Write a function which given a list of integers and an integer, will return all integers smaller than or equal to that integer.
Example: `filter([1,2,3,4,5], 3)` ⇒ `[1,2,3].`

B. Write a function which given a lists will reverse the order of the elements.
Example: `reverse([1,2,3])` ⇒ `[3,2,1].`

C. Write a function which, given a list of lists, will concatenate them.
Example: `concatenate([[1,2,3], [], [4, five]])` ⇒ `[1,2,3,4,five].`
Hint: You will have to use a help function and concatenate the lists in several steps.

D. Write a function which given a list of nested lists, will return a flat list.
Example: `flatten([[1,[2,[3],[]]], [[[4]]], [5,6]])` ⇒ `[1,2,3,4,5,6].`
Hint: use concatenate

Advanced Exercise 6: Implement Quicksort and Merge sort

Implement the following algorithms over lists:

Quicksort
The head of the list is taken as the pivot; the list is then split according to those elements smaller than the pivot and the rest. These two lists are then recursively sorted by quicksort and joined together with the pivot between them.

Merge sort
The list is split into two lists of (almost) equal length. These are then sorted separately and their result merged together.

Concurrent Programming

These exercises will help you get familiar with the syntax and semantics of concurrency in Erlang. You will solve problems that deal with spawning processes, message passing, registering, and termination. If you are having problems finding bugs or following what is going on, use the process manager.

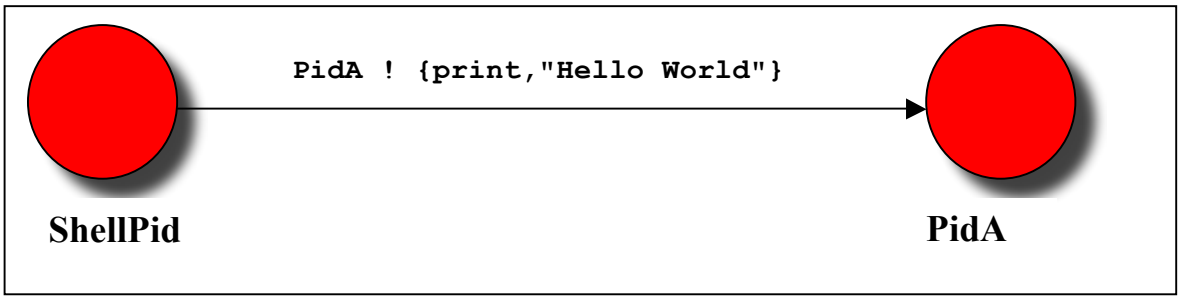
Exercise 1: An Echo Server

Write a server which will wait in a receive loop until a message is sent to it. Depending on the message, it should either print it and loop again or terminate. You want to hide the fact that you are dealing with a process, and access its services through a functional interface. These functions will spawn the process and send messages to it. The module echo.erl should export the following functions.

Interface:

```
echo:start() ⇒ ok.
echo:stop() ⇒ ok.
echo:print(Term) ⇒ ok.
```

Example:



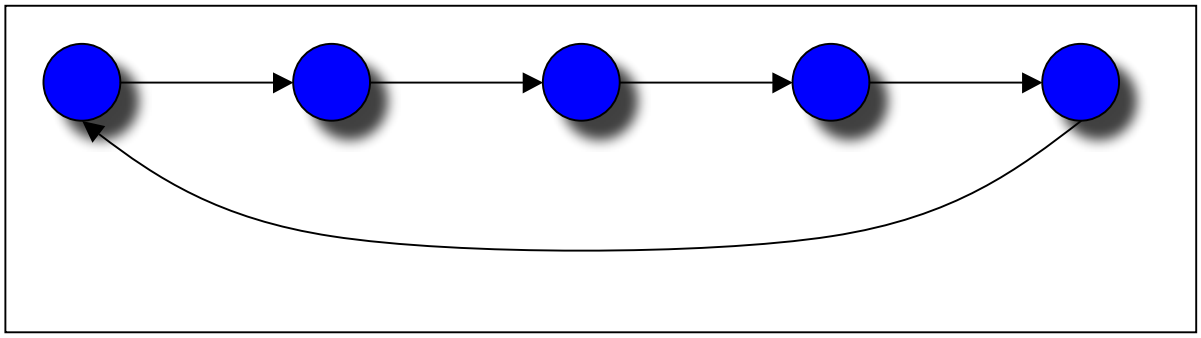
Hint: Use the register/2 built in function.

Warning: Use an internal message protocol to avoid stopping the process when you for example call the `function echo:print(stop).`

Exercise 2: The Process Ring

Write a program that will create N processes connected in a ring. These processes will then send M number of messages around the ring and then terminate gracefully when they receive a quit message.

Example

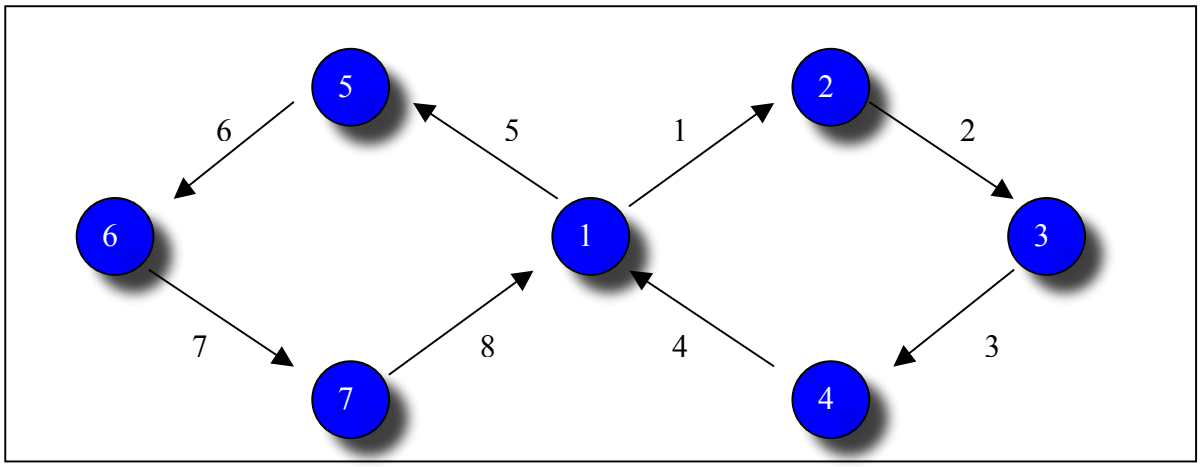


Hint: There are two basic strategies to tackling your problem. The first one is to have a central process that sets up the ring and initiates the message sending. The second strategy consists of the new process spawning the next process in the ring. With this strategy you have to find a method to connect the first process to the last.

Exercise 3: The Process Crossring

Write a program that will create N processes connected in a ring. These processes will then send M number of messages around the ring. Halfway through the ring, however, the message will cross over the first process, which will then forward it to the second half of the ring. The ring should terminate gracefully when receiving a quit message.

Example



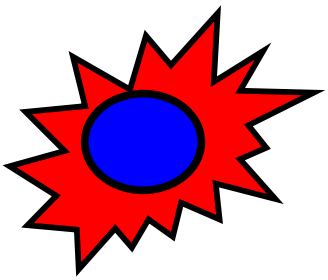
Example:

```
1> crossring:start(7,2,hello).
Process: 2 received: hello
<0.618.0>
Process: 3 received: hello
Process: 4 received: hello
Process: 1 received: hello halfway through
Process: 5 received: hello
Process: 6 received: hello
Process: 7 received: hello
Process: 1 received: hello
Process: 2 received: hello
Process: 3 received: hello
Process: 4 received: hello
Process: 1 received: hello halfway through
Process: 5 received: hello
Process: 6 received: hello
Process: 7 received: hello
Process: 1 received: hello
Process: 1 terminating
Process: 2 terminating
Process: 5 terminating
Process: 3 terminating
Process: 6 terminating
Process: 4 terminating
Process: 7 terminating
```

Process Error Handling

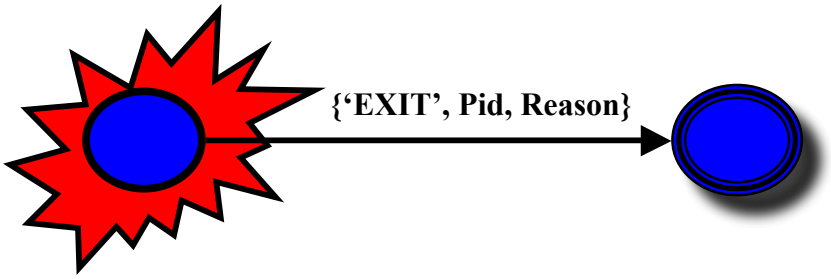
The aim of these exercises is to make you practice the simple but powerful error handling mechanisms found in Erlang. They include exiting, linking, trapping of exits and the use of catch.

Exercise 1: The Linked Ping Pong Server



Modify the processes A and B from the file **pingpong.erl** in the exercises files given to you, by linking the processes to each other. When the stop function has been called, instead of sending a quit message, make the first process terminate abnormally. This should result in the exit signal propagating to the other process, causing it to terminate as well.

Exercise 2: Trapping Exits



The file **fussball.erl** contains a program similar to the ping pong program, i.e. it sends a message back and forth between two processes, only in a slightly more entertaining fashion.

An example of how to run the code follows:

```
47> fussball:start(germany,france).
ok
48> fussball:start(france, germany).
ok
49> fussball:kickoff(germany).
ok
germany kicks the ball...
france kicks the ball...
germany kicks the ball...
france kicks the ball...
germany kicks the ball...
france kicks the ball...
germany SCORES!!
Oh no! germany just scored!!
50> fussball:stop(france).
stop
51> fussball:stop(germany).
```

```
stop
```

Modify the code from the Fussball exercise to make the processes trap exits. Do this by inserting the following line first in the **init** function:

```
process_flag(trap_exit, true),
```

Exit signals will now be added to the message queue instead of terminating the processes. To make the processes print out the exit signals they receive, add the following **receive** clause to the **loop** function:

```
{'EXIT', _Pid, Reason} ->
        io:format("Got exit signal: ~p~n", [Reason]);
```

Find a way to link both countries together in the init phase so that whenever one of the countries is stopped, the other also is. **Hint**: linking to a non-existing process causes an exception. You should handle that.

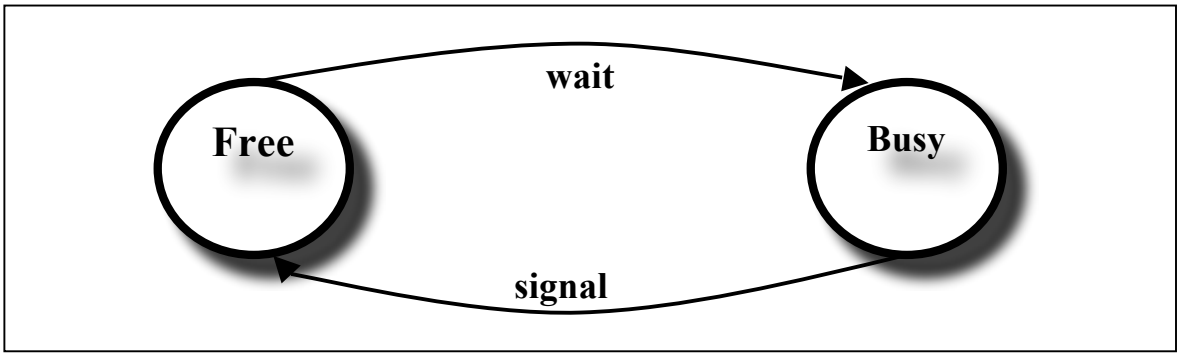
Exercise 3: A Reliable Mutex Semaphore

The file `mutex.erl` contains a binary semaphore module, providing mutual exclusion (mutex) for processes that want to share a resource. The mutex is modeled as a finite state machine with two states, `busy` and `free`. If a process tries to take the mutex (by calling **mutex:wait()**) when the process is in state busy, the function call hangs until the mutex becomes available (namely, the process holding the mutex calls **mutex:signal()**).

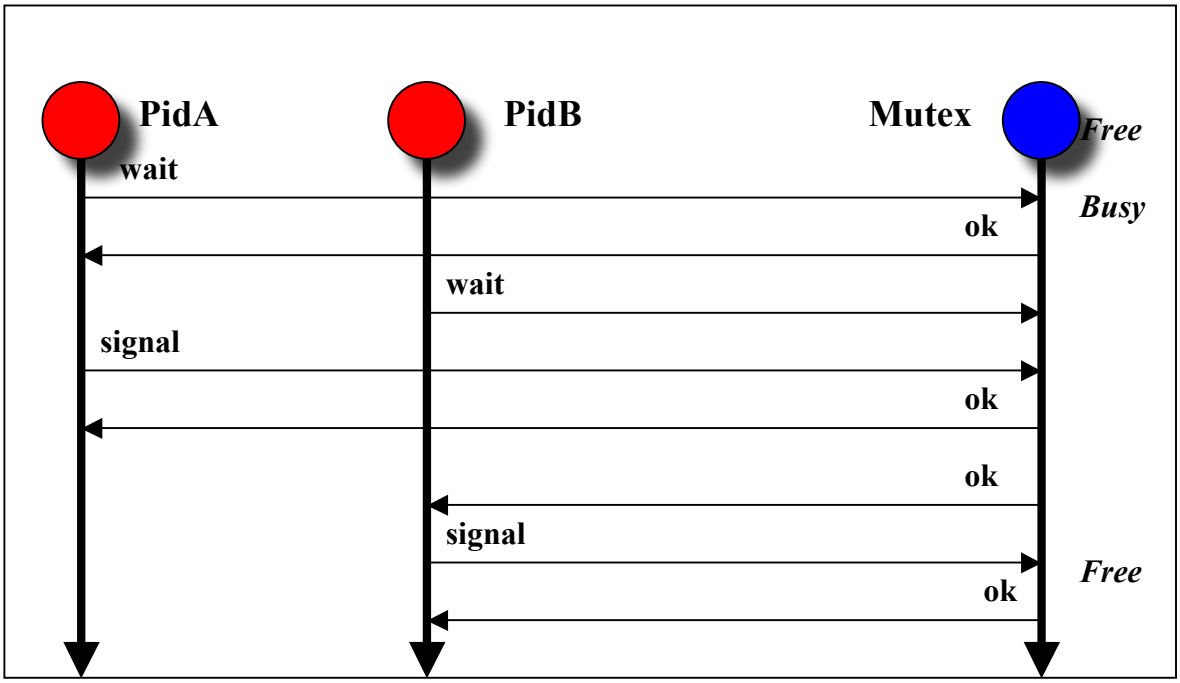
Interface

mutex:start() ⇒ ok.
mutex:wait() ⇒ ok.
mutex:signal() ⇒ ok.

Finite State Machine



Message Sequence Chart



The mutex semaphore is unreliable. What happens if a process that currently holds the semaphore terminates prior to releasing it? Or what happens if a process waiting to execute is terminated due to an exit signal? By trapping exits and linking to the process that currently holds the semaphore, make your mutex semaphore reliable.

Hint: Use **catch link(Pid)** in case Pid terminated before its request was handled.

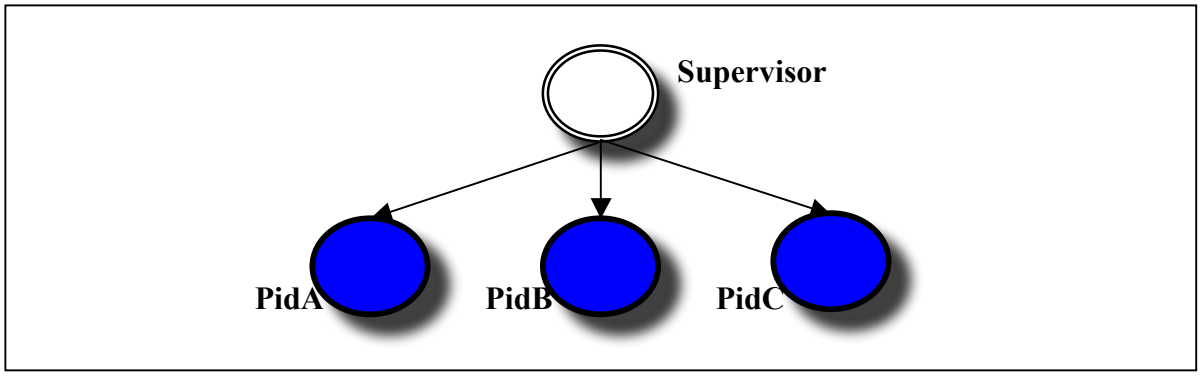
Advanced Exercise 4: A Supervisor Process

Write a supervisor process that will spawn children and monitor them. If a child terminates abnormally, it will print an error message and restart it. To avoid infinite restarts (What if the Module did not exist?), put a counter which will restart a child a maximum of 5 times, and print an error message when it gives up and removes the child from its list. Stopping the supervisor should unconditionally kill all the children.

Interface

```
sup:start(SupName) ⇒ {ok, Pid}.
sup:start_child(SupName | Pid, Mod, Func, Args) ⇒ {ok, Pid}.
sup:stop(SupName | Pid) ⇒ ok.
```

Example 1



Example 2

```
1> c(sup).
{ok,sup}
2>
2> sup:start(freddy).
{ok,<0.41.0>}
3> {ok, Pid} = sup:start_child(freddy, my_db, init, []).
{ok,<0.43.0>}
4> exit(Pid, kill).
true
---------------------------------------------
Error: Process <0.43.0> Terminated 1 time(s)
      Reason for termination:killed
      Restarting with my_db:init/0
---------------------------------------------
5> {ok, Pid2} = sup:start_child(freddy, my_db, init, []).
{ok,<0.47.0>}
6> i().
                                        [snip]
```

Hint: Make your supervisor start the mutex and database server processes. Note that you have to pass the function and arguments used in the spawn function, and not the start function. That might result in your process not getting registered.

If it is getting registered, kill it by using **exit(whereis(ProcName), kill).** See if they have been restarted by calling **whereis(ProcName)** and ensuring you are getting different Process Ids every time.

If the process is not registered, kill it by calling **exit(Pid, kill).** You will get Pid from the return value of the start_child function. (You can then start many processes of the same type). Once killed, check if the process has been restarted by calling the **i().** help function. It lists all the processes in the system, their initial function and the current function they are executing in.

Note: SupName | Pid means you should be able to pass either the atom by with the supervisor process is registered by, or the Process ID returned when the supervisor is started.