

1 Starting the System and Basic Erlang

These exercises will help you get accustomed with the Erlang development and run time environments. The exercises marked **Advanced** are harder or more complex exercises to push your understanding of Erlang further. 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).

```
_____ Unix Shell _____  
$ erl  
  
_____ Erlang Shell Session _____  
Eshell V5.9.1 (abort with ^ G)  
1>
```

1.1 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.
```

1.2 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.

1.3 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?

```
----- Erlang Shell Session -----  
Eshell V5.9.1 (abort with ^ G)  
1> c(demo).  
{ok, demo}  
2> demo:double(6).  
12
```

1.4 Temperature Conversion

Part A

Write functions `temp:f2c(Fahrenheit)` and `temp:c2f(Celsius)` which convert between Fahrenheit and Celsius temperature scales in a module named *temp.erl*.

Hint:

$$5 * (F - 32) = 9 * C$$

Part B

Write a function `temp:convert(Temperature)` which combines the functionality of `f2c` and `c2f`. A usage example follows:

```
Eshell V5.9.1 (abort with ^ G)
1> temp:convert({c, 100}).
{f, 212.0}.
2> temp:convert({f, 32}).
{c, 0.0}.
```

1.5 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

Table 1: Boolean Operators

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.

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

Note:

$\text{foo}(X) \rightarrow Y$ means that calling the function `foo` with the parameter `X` will result in the value `Y` being returned.

Note:

`and`, `or` and `not` are reserved words in Erlang.

2 Sequential Programming

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

2.1 Evaluating Expressions

Part A

Write a function `sum/1` which given a positive integer N will return the sum of all the integers between 1 and N .

```
sum(5) → 15.
```

Part B

Write a function `sum_interval/2` which given two integers N and M , where $N \leq M$, will return the sum of the interval between N and M . If $N > M$, you want your process to terminate abnormally.

```
sum_interval(1,3) → 6.  
sum_interval(6,6) → 6.
```

2.2 Creating Lists

Part A

Write a function which returns a list of the format $[1, 2, \dots, N-1, N]$.

```
create(3) → [1, 2, 3].
```

Part B

Write a function which returns a list of the format $[N, N-1, \dots, 2, 1]$.

```
reverse_create(3) → [3, 2, 1].
```

2.3 Side Effects

Part A

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

```
————— Erlang Shell Session —————  
Eshell V5.9.1 (abort with ^ G)  
1> print(5).  
1  
2  
3  
4  
5  
ok
```

Hint:

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

Part B

Write a function which prints out the even integers between *1* and *N*.

```
_____ Erlang Shell Session _____  
Eshell V5.9.1 (abort with ^ G)  
1> even_print(5).  
2  
4  
ok
```

Hint:

Use guards

2.4 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.

```
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].
```

```

Eshell V5.9.1 (abort with ^ G)
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.

2.5 ADVANCED: Manipulating Lists

Part A

Write a function which given a list of integers and an integer, will return all integers smaller than or equal to that integer.

```
filter([1,2,3,4,5], 3) → [1,2,3].
```

Part B

Write a function which given a lists will reverse the order of the elements.

```
reverse([1,2,3]) → [3,2,1].
```

Part C

Write a function which, given a list of lists, will concatenate them.

```
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.

Part D

Write a function which given a list of nested lists, will return a flat list.

```
flatten([[1, [2, [3], []]], [[4]], [5, 6]]) → [1, 2, 3, 4, 5, 6].
```

Hint:

Use the concatenate function.

2.6 ADVANCED: 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.

Mergesort

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

3 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.

3.1 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.

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

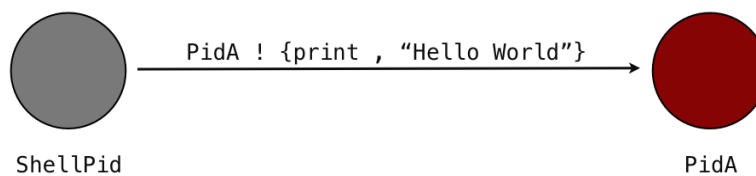


Figure 1: An Echo Server

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)`.

3.2 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.

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.

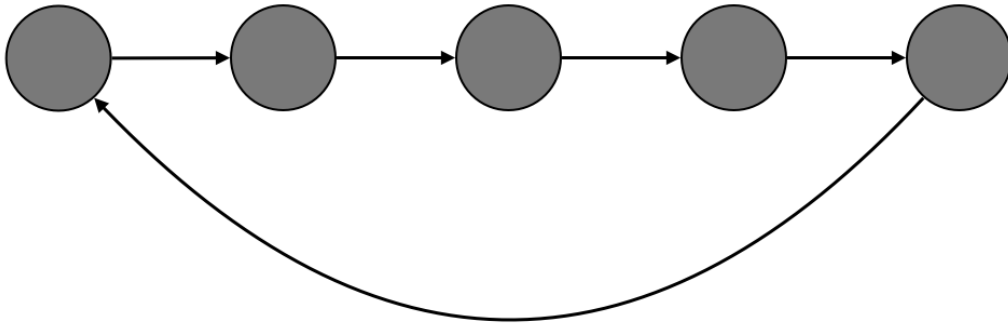


Figure 2: The Process Ring

3.3 The Process Crossing

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.

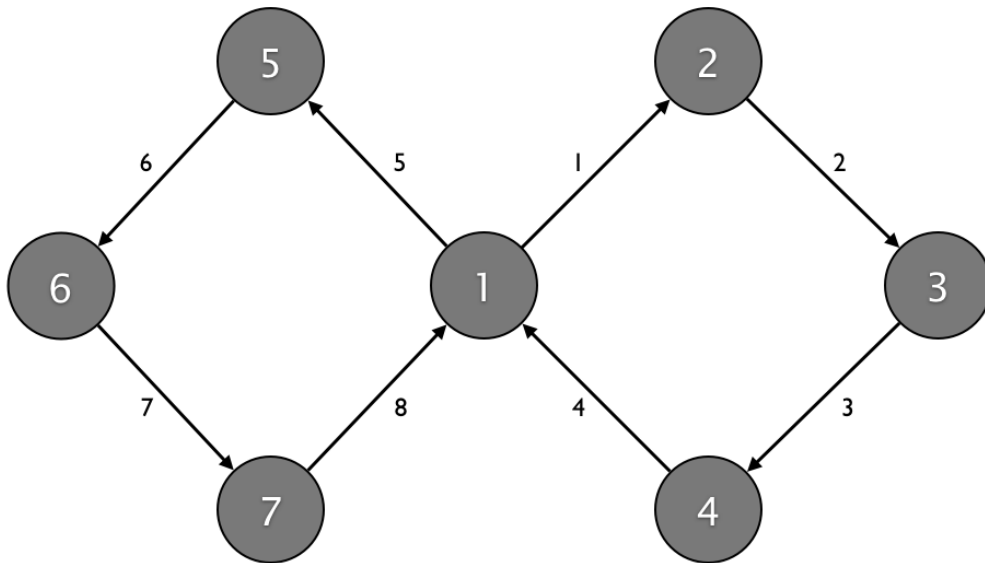


Figure 3: The Process Crossing

```
Eshell V5.9.1 (abort with ^ G)
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
```

4 Process Design Patterns

These exercises will help you get familiar with process design patterns. Similar patterns will occur in different programs, and are the building blocks of systems based on OTP. Understanding them and knowing when to use which pattern is crucial to keeping the code simple and clean.

4.1 A Database Server

Write a database server that stores a database in its loop data. You should register the server and access its services through a functional interface. Exported functions in the `my_db.erl` module should include:

```
my_db:start() → ok.  
my_db:stop() → ok.  
my_db:write(Key, Element) → ok.  
my_db:delete(Key) → ok.  
my_db:read(Key) → {ok, Element} | {error, instance}.  
my_db:match(Element) → [Key1, ..., KeyN].
```

Hint:

Use the `db.erl` module as a back end and use the server skeleton from the echo exercise.

————— Erlang Shell Session —————

```
Eshell V5.9.1 (abort with ^ G)  
1> my_db:start().  
ok  
2> my_db:write(foo, bar).  
ok  
3> my_db:read(baz).  
{error, instance}  
4> my_db:read(foo).  
{ok, bar}  
5> my_db:match(bar).  
[foo]
```

4.2 A Mutex Semaphore

Write a process that will act as a binary semaphore providing mutual exclusion (mutex) for processes that want to share a resource. Model your process 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 should hang until the mutex becomes available (namely, the process holding the mutex calls `mutex:signal()`).

```
mutex:start() → ok.  
mutex:wait() → ok.  
mutex:signal() → ok.
```

Hint:

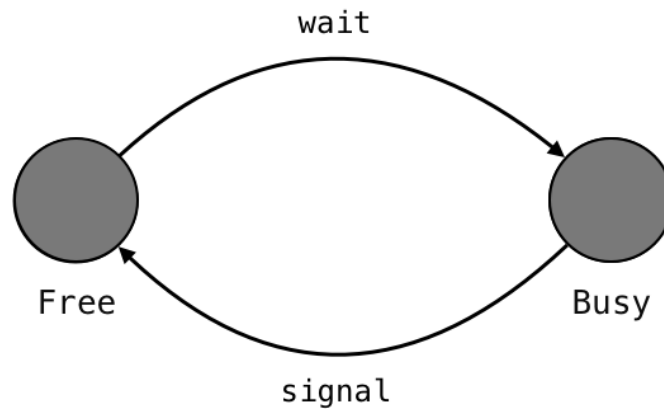


Figure 4: The Mutex Finite State Machine

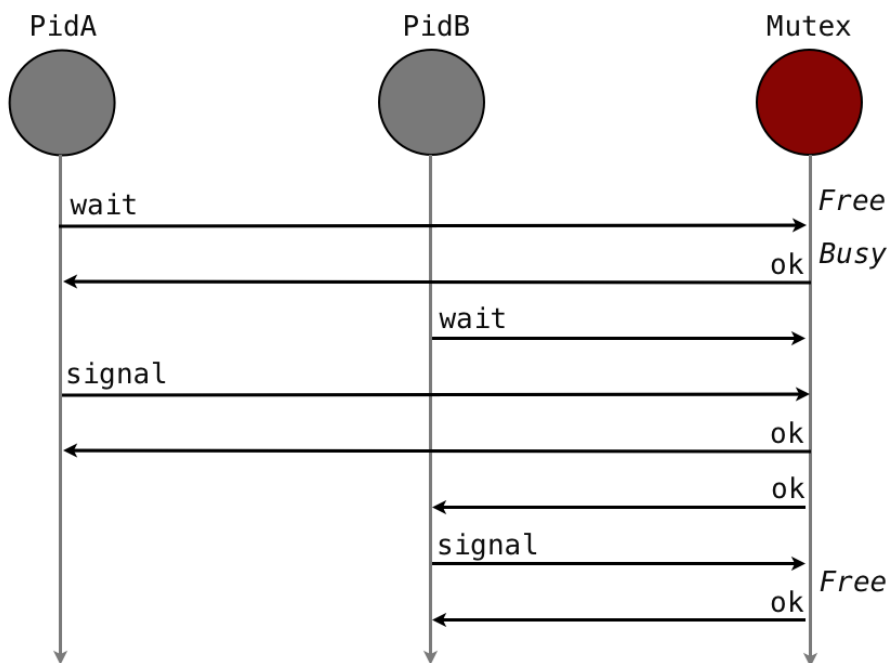


Figure 5: The Mutex Message Sequence Chart

The difference in the state of your FSM is which messages you handle in which state.

4.3 ADVANCED: A Database Server with transactions

Rewrite the Database server from Exercise 4.1 to add transactions. This should be done with exported functions:

```

my_db:lock() → ok.
my_db:unlock() → ok.

```

A client starts a transaction by locking the server by calling `my_db:lock` and ends it by calling `my_db:unlock`. During the transaction the server will block requests from other clients and process them first **after** the transaction has ended.

5 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.

5.1 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.

5.2 Trapping Exits

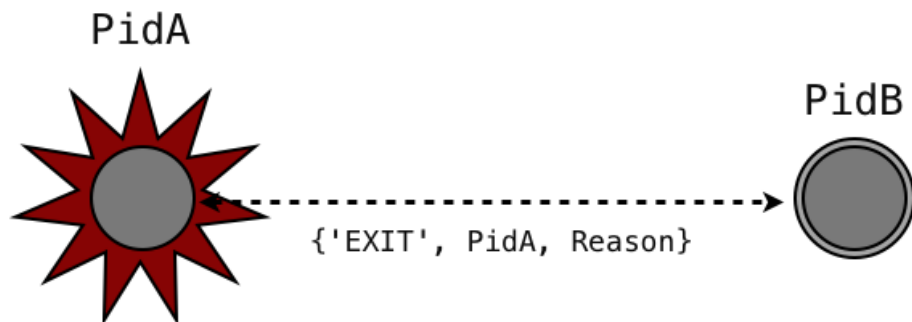


Figure 6: Trapping Exits

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:

```
Eshell V5.9.1 (abort with ^ G)
1> fussball:start(germany,france).
ok
2> fussball:start(france, germany).
ok
3> 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!!
4> fussball:stop(france).
stop
5> 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.

5.3 A Reliable Database Server with transactions

Your Database server from Exercise 4.3 is unreliable. What happens if the process locking the server terminates prior to unlocking it? By trapping exits and linking to the locking process make the database server reliable.

5.4 A Reliable Mutex Semaphore

Your Mutex semaphore from exercise 4.2 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.

5.5 ADVANCED: 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.

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

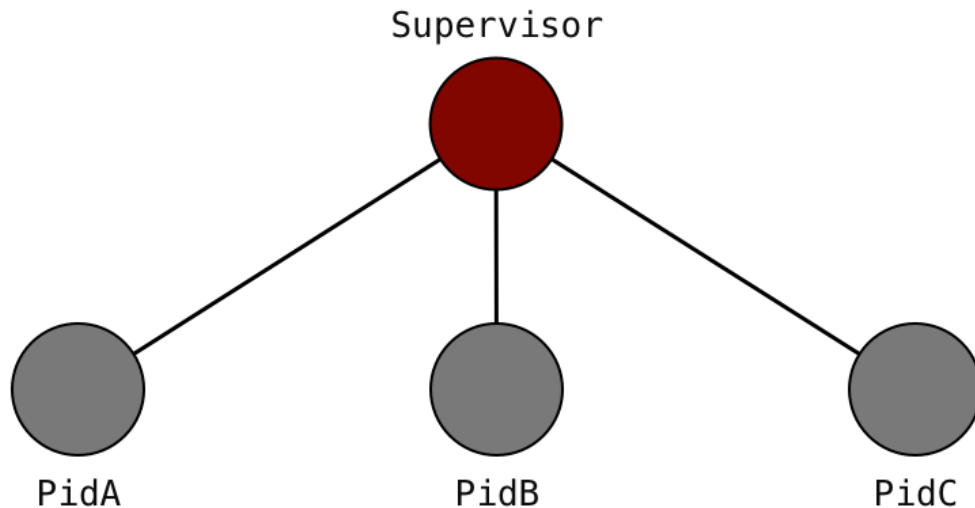


Figure 7: A supervisor Process

```

----- Erlang Shell Session -----
Eshell V5.9.1 (abort with ^ G)
1> c(sup).
{ok,sup}
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().

```

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 which the supervisor process is registered by, or the Process ID returned when the supervisor is started.

6 Records and Funs

6.1 Database Handling Using Records

Take the `db.erl` module you wrote in exercise 2.4. Rewrite it using records. Test it using your database server you wrote in exercise 4.1. As a record, you could use the following definition. Remember to place it in an include file.

Hint:

Use the following record definition: `-record(data, {key, data}).`

Note:

Make sure you save a copy of your `db.erl` module using lists somewhere else (or with a new name) before you start changing in.

6.2 Higher Order Functions

Part A

Using funs and higher order functions, write a function which prints out the integers between 1 and N .

Hint:

Use `lists:seq(1, N).`

Part B

Using funs and higher order functions, write a function which given a list of integers and an integer, will return all integers smaller than or equal to that integer.

Part C

Using funs and higher order functions, write a function which prints out the even integers between 1 and N .

Hint:

Solve your problem either in two steps, or use two clauses in your fun.

Part D

Using funs and higher order functions, write a function which, given a list of lists, will concatenate them.

Part E

Using funs and higher order functions, write a function that given a list of integers returns the sum of the integers.

Hint:

Use `lists:foldl`, and try figure out why we prefer to use `foldl` rather than `foldr`.

7 Advanced Topics

These exercises will make you familiar with issues covered in the advanced topics section. In many cases, you will be using exercises created in the previous section, making the programs more robust or adding to their functionality.

7.1 ADVANCED: List Comprehensions

Part A

Using List comprehensions, create a set of integers between 1 and 10 which are divisible by three.

Example:

```
[3, 6, 9]
```

Part B

Using list comprehensions remove all non integers from a polymorphic list. Return the list of integers squared.

Example:

```
[1, hello, 100, "boo", 9] should return [1, 10000, 81]
```

Part C

Using list comprehensions and given two lists, return a new list which is the intersection of the two lists.

Example:

```
[1, 2, 3, 4, 5] and [4, 5, 6, 7, 8] should return [4, 5]
```

Hint:

Assume that the lists contain no duplicates.

Part D

Using list comprehensions and given two lists, return a new list which is the disjunction of the two lists.

Example:

```
[1, 2, 3, 4, 5] and [4, 5, 6, 7, 8] should return [1, 2, 3, 6, 7, 8]
```

Hint:

Assume that the lists contain no duplicates.

7.2 Database Handling using ETS

Take the `db.erl` module from exercise 6.1, and rewrite it using ETS tables instead of operations on lists. Use records to store your data in the ets tables. If you are having problems, use the table visualizer tool to debug your code.

Test your back end module using your database server you wrote in exercise 4.1.

Note:

Just in case you still need that reminder. Make sure you save a copy of your `db.erl` module using lists somewhere else (or with a new name) before you start changing in.

7.3 ADVANCED: Software Upgrade During Run Time

Part A

Take the `db.erl` module you wrote using ETS and add an extra function called `code_upgrade/1`. This function will take a database of the format used in exercise 6.1. It should create an ETS table and store all the elements passed to it in the table. Its return value should return the ETS table reference.

```
db:code_upgrade([RecordList]) → TabRef.
```

Part B

Test this function in the shell, and place the beam (and source code) files a sub directory called *patches*. For section C, continue working in your current working directory where you have stored the `db.erl` module using lists. Be careful with the different compiled versions of the two modules so that you do not mix them up.

Part C

Add the function `code_upgrade(OldDb)` to `my_db.erl` server module. This function should send a message to the server which will load the new `db` module, call the `db:code_upgrade/1` function with the old database, and then continue looping with the new database representation. After the code upgrade, all the data that was there prior to the operation should still be available, and the client should be able to insert, delete, and do queries on elements as if nothing has happened.

Part D

Test your program by starting it up using the lists version of the `db` module. Insert a few elements in the database, and switch over to the ETS version of the `db` module. To switch over to the new version, you must first load it in the system. (In larger systems, there will be tools to handle upgrades. As our database server is of more modest proportions, we will load it manually). Once the ETS version of the `db.erl` module has been loaded, call `my_db:code_upgrade()`. Read the elements, write some new ones, and delete the old ones. Make sure the server is stable.

Hint:

Look at the code manual page to find out how to load new versions of modules during run time and how to add code search paths.

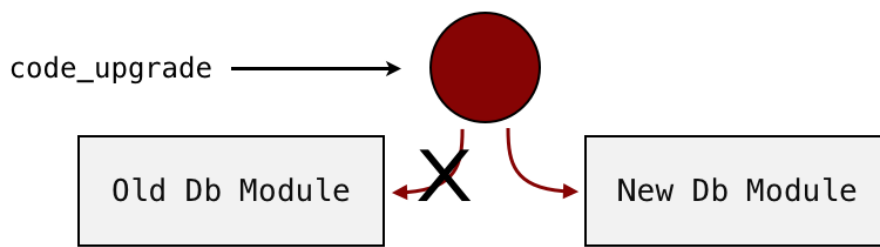


Figure 8: Runtime Upgrade

```
----- Erlang Shell Session -----
Eshell V5.9.1 (abort with ^ G)
1> my_db:start().
ok
2> my_db:write(bob, handyman).
ok
3> my_db:write(chris, projectleader).
ok
4> my_db:read(bob).
{ok, handyman}
5> code:add_patha("/home/cesarini/erlang/course/patches/").
true
6> my_db:code_upgrade().
ok
7> my_db:read(bob).
{ok, handyman}
8> my_db:delete(bob).
ok
9> my_db:write(bob, manager).
ok
10> code:soft_purge(db).
true
```

7.4 Distribution

Make your server from exercise 7.2 (*Database Handling using ETS*) a distributed one. Clients should, through the functional interface, be able to query the node from any distributed Erlang node sharing the same cookie. Using a macro, you may hardcode the node name of the server.

Sending queries on an open network means there is a risk of interference from a third party. Add references in the message protocol between the client and the server, to ensure that you are in fact receiving the reply to that query, and not just any message following the protocol.

The node on which the server is running may be down or may crash just after you have sent your request. Ensure that these cases are handled, and that you return an error should they occur.

7.5 Distributed, Replicated Associative Store

Design a distributed version of an associative store in which values are associated with tags. It is possible to store a tag/value pair, and to look up the value(s) associated with a tag. One example for this is an address book for email, in which email addresses (values) are associated with nicknames (tags).

Replicate the store across two nodes on the same host, send lookups to one of the nodes (chosen either at random or alternately), and send the updates to both. How could you re-implement the system to include three or four store nodes? Design a system to test your answer to this exercise.

8 Behaviours

These exercises will help you get familiar with process design patterns. Similar patterns will occur in different programs, and are the building blocks of systems based on OTP. Understanding them and knowing when to use which pattern is crucial to keeping the code simple and clean.

8.1 A Database Server with transactions

Rewrite the Database server either from Exercise 4.1 or 4.3 to make it an OTP `gen_server`. It should be tested with the same test modules as for the original servers.

8.2 A Database Server Supervisor

Write an OTP supervisor for the Database server from 8.1. How would the Database server need to be modified to handle restarts without losing data?

8.3 A Persistent Database Server with dets

Part A

Rewrite the `db.erl` module from 2.4 or 6.1 to use DETS instead of operations on lists. Change the `new/0` function to take an argument which is the name of the file to use.

Hint:

Make sure you save a copy of your `db.erl` module using lists somewhere else (or with a new name) before you start changing in. Or you may give it another name.

Hint:

A DETS table is created/opened with the function `dets:open_file(FileName)` not the function `new`.

Part B

Modify the Database server from Exercise 8.1 to use the new DETS based `db.erl` module. It should be tested with the same test modules as for the original servers.