Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-322 — Sistemas Operativos I

Práctica 5

Erlang

Introducción a Erlang

Erlang es un lenguaje de programación concurrente y un sistema de ejecución que incluye una máquina virtual y bibliotecas. La creación y gestión de procesos es trivial (y barata) en Erlang, mientras que, en muchos lenguajes, los hilos se consideran un apartado complicado y propenso a errores. En Erlang toda concurrencia es explícita. Los programas se guardan en archivos con extensión `.erl` y se los compila desde el shell de Erlang `erl` con la función `c/1`. Los módulos deben exportar alguna función (con la cláusula `-export`) para poder invocarla desde el shell. Cada *statement* debe terminar con un punto. Para salir del intérprete, puede llamar a la función `q()`.

El shell de Erlang también es un proceso; se puede usar `self()`. para obtener el pid actual.

Ej. 1 (Intro). Dado el siguiente código:

```
-module(intro).
-export([init/0]).

match_test () ->
  {A,B} = {5,4},
  {C,C} = {5,5},
  {B,A} = {4,5},
  {D,D} = {5,5}.

string_test () -> [
  helloworld == 'helloworld',   %true
  "helloworld" < 'helloworld',  %false
  helloworld == "helloworld",    %false
  [$h,$e,$l,$l,$o,$w,$o,$r,$l,$d] == "helloworld",  %true
  [104,101,108,108,111,119,111,114,108,100] < {104,101,108,108,111,119,111,114,108,100},  %flase
  [104,101,108,108,111,119,111,114,108,100] > 1,   %true
  [104,101,108,108,111,119,111,114,108,100] == "helloworld]. %true

tuple_test (P1, P2) ->
  io:fwrite("El nombre de P1 es ~p y el apellido de P2 es ~p~n", [nombre(P1), apellido(P2)]).

apellido (P) -> ok.
nombre (P) -> ok.

filtrar_por_apellido(Personas, Apellido) -> ok.

init () ->
  P1 = {persona, {nombre, "Juan"}, {apellido, "Gomez"}},
  P2 = {persona, {nombre, "Carlos"}, {apellido, "Garcia"}},
  P3 = {persona, {nombre, "Javier"}, {apellido, "Garcia"}},
```

```
P4 = {persona, {nombre, "Rolando"}, {apellido, "Garcia"}},
match_test(),
tuple_test(P1, P2),
string_test(),
Garcias = filtrar_por_apellido([P4, P3, P2, P1], "Garcia").
```

a) Justifique cuáles *match* de la función `match_test` deberían ser válidos y cuáles no.

b) Implemente las funciones `nombre/1` y `apellido/1` para que devuelvan esos campos de las tuplas que obtienen como argumento utilizando pattern matching.

c) Explique el resultado de cada una de las comparaciones de la función `string_test/0` (es decir por qué dan `true` o `false`).

d) Implemente la función `filtrar_por_apellido/2` para que devuelva los nombres (sin el apellido) de las personas de la lista `Personas` cuyo apellido coincide con `Apellido` utilizando comprensión de listas.

Ej. 2 (Temporización).

a) Implemente una función `wait/1` que tome como argumento una cantidad de milisegundos y espere ese tiempo.

b) Implemente un cronómetro que reciba tres argumentos, `Fun`, `Hasta` y `Periodo` y ejecute `Fun/0` cada `Periodo` milisegundos hasta que hayan pasado `Hasta` milisegundos **sin bloquear el intérprete**. Un caso de prueba sería:

```
cronometro(fun () -> io:fwrite("Tick~n") end, 60000, 5000).
```

que imprimiría `Tick` cada 5 segundos durante un minuto.

Ej. 3 (Balanceo de Carga). La siguiente función implementa un *balanceo de carga* sobre una lista de servidores. Al recibir un pedido de la forma `{req, Arg, Pid}`, lo reenvía aleatoriamente a uno de los servidores y espera su respuesta. Luego, reenvía la respuesta al cliente que originó el pedido.

```
bal(Servs) ->
    receive
        {req, Arg, Pid} ->
            Proc = lists:nth(rand:uniform(length(Servs)), Servs),
            Proc ! { req, Arg, self() },
            receive
                Reply -> Pid ! Reply
            end
    end,
    bal(Servs).
```

Explique el problema con esta implementación. Proponga una solución e impleméntela.

Ej. 4 (Servidor de Turnos). Reimplemente el servidor de tickets de la Práctica 2 en Erlang. Puede usar el siguiente esqueleto para manejar conexiones TCP en Erlang. El mismo acepta conexiones TCP en *modo activo*, haciendo que el proceso que realizar el `accept` de una conexión reciba mensajes con los datos recibidos por la misma. También puede usar el *modo pasivo* si así lo desea, cambiando las

opciones pasadas a `listen`. Asegúrese también de que el servidor es robusto: debe manejar correctamente conexiones cerradas por el cliente y también tener en cuenta que los pedidos pueden llegar fragmentados o "pegados" (TCP no tiene concepto de mensaje ni de "borde"), entre otras cosas.

```
-module(turnos).
-export([server/0]).

server() ->
        {ok, ListenSocket} = gen_tcp:listen(8000, [{reuseaddr, true}]),
        wait_connect(ListenSocket, 0).

wait_connect(ListenSocket, N) ->
        {ok, Socket} = gen_tcp:accept(ListenSocket),
        spawn (fun () -> wait_connect (ListenSocket, N+1) end),
        get_request(Socket).

get_request(Socket) ->
        io:fwrite("Esperando mensajes de ~p~n", [Socket]),
        receive
                _X -> ok,
                      get_request(Socket)
        end.
```

a) Compare el servidor en PThreads y el actual con el cliente dado anteriormente, para 200, 2000 y 20000 conexiones simultáneas. Puede usar el cliente `turno_cliente.c`.

b) ¿Ve una diferencia importante en el consumo de memoria de los dos servidores? ¿A qué cree que se puede deber?

c) ¿Puede cada servidor aceptar 50000 conexiones simultáneas?

Nota: para conseguir aceptar tantas conexiones, seguramente tenga que aumentar el `ulimit` de FDs abiertos que impone el sistema operativo. Correr `ulimit -n 1000000` debería bastar. Ver también `help ulimit`.

Ej. 5 (Lanzar Procesos en Anillos). Escriba un programa que lance N procesos en anillos. Cada proceso recibirá dos clases de mensajes:

- `{msg, N}` donde `N` es un entero. Deberá decrementarlo y enviarlo al siguiente proceso en el anillo si `N` es mayor que cero. En caso contrario deberá enviar un mensaje `exit` y terminar cuando todos los demás lo hayan hecho.
- `exit` cuando el proceso debe terminar.

Modifique el programa para que el mensaje enviado gire una vez alrededor del anillo y sea descartado por el que inició el envío.

Ej. 6 (Suma y Consenso). Reimplementar la suma por consenso de la Práctica 4 en Erlang.

Compare esta nueva implementación con la de MPI:

a) ¿Qué similitudes y diferencias encontró en cuanto a la implementación?

b) Cómpare la performance y la robustez. ¿Qué pasa si un proceso muere en cada caso?

Nota: En linux puede usar el comando `/usr/bin/time -v PROGRAMA` para ver el uso de memoria. Por ejemplo haciendo `/usr/bin/time -v erl` y luego saliedo del interprete podemos tener una noción de cuanta memoria usa Erlang para levantar su entorno de ejecución.

Para un profiling más detallado se puede usar algunas de las herramientas que provee Erlang para profiling[1].

Ej. 7 (Servidor de Difusión). Implemente un proceso servidor que distribuya los mensajes que recibe entre todos sus suscriptores. El servidor tiene las siguientes operaciones:

Suscribirse:	El proceso llamado es incluido en el conjunto de suscriptores.
Enviar mensaje:	El mensaje recibido debe ser reenviado a todos los suscriptores.
Desuscribirse:	El proceso llamado es eliminado del conjunto de suscriptores.

Cada operación debe tener una función que la implemente, por ejemplo `suscribir/1`. El servidor puede iniciarse con una función que retorne un descriptor del mismo, o puede registrarse globalmente (en cuyo caso, `suscribir` tiene aridad cero). Si hay paso de mensajes, el mismo debe estar abstraído detrás de esa interfaz. En cada difusión, los suscriptores deben recibir el mensaje una única vez. Una vez suscrito, no debería tener efecto suscribirse nuevamente, y desuscribirse siempre tiene efecto inmediato (i.e. las suscripciones no son recursivas).

Ej. 8 (Sincronización). Complete el código siguiente para implementar Locks y Semáforos en Erlang usando paso de mensajes. Las funciones `testLock/0` y `testSem/0` son casos de uso.

```
-module(sync).
-export([createLock/0, lock/1, unlock/1, destroyLock/1]).
-export([createSem/1, semP/1, semV/1, destroySem/1]).
-export([testLock/0, testSem/0]).

createLock () -> throw (undefined).
lock (_L) -> throw (undefined).
unlock (_L) -> throw (undefined).
destroyLock (_L) -> throw (undefined).

createSem (_N) -> throw(undefined).
destroySem (_S) -> throw (undefined).
semP (_S) -> throw (undefined).
semV (_S) -> throw (undefined).

f (L, W) ->
        lock(L),
        % regioncritica(),
        io:format("uno ~p~n", [self()]),
        io:format("dos ~p~n", [self()]),
        io:format("tre ~p~n", [self()]),
        io:format("cua ~p~n", [self()]),
        unlock(L),
        W ! finished.

waiter (L, 0) -> destroyLock(L);
waiter (L, N) -> receive finished -> waiter(L, N-1) end.

waiter_sem (S, 0) -> destroySem(S);
waiter_sem (S, N) -> receive finished -> waiter_sem(S, N-1) end.

testLock () ->
        L = createLock(),
```

[1] https://www.erlang.org/docs/22/efficiency_guide/profiling.html

```
        W = spawn(fun () -> waiter(L, 3) end),
        spawn (fun () -> f(L, W) end),
        spawn (fun () -> f(L, W) end),
        spawn (fun () -> f(L, W) end),
        ok.

sem (S, W) ->
        semP(S),
        %regioncritica(), bueno, casi....
        io:format("uno ~p~n", [self()]),
        io:format("dos ~p~n", [self()]),
        io:format("tre ~p~n", [self()]),
        io:format("cua ~p~n", [self()]),
        io:format("cin ~p~n", [self()]),
        io:format("sei ~p~n", [self()]),
        semV(S),
        W ! finished.

testSem () ->
        S = createSem(2), % a lo sumo dos usando io al mismo tiempo
        W = spawn (fun () -> waiter_sem (S, 5) end),
        spawn (fun () -> sem (S, W) end),
        spawn (fun () -> sem (S, W) end),
        spawn (fun () -> sem (S, W) end),
        spawn (fun () -> sem (S, W) end),
        spawn (fun () -> sem (S, W) end),
        ok.
```

De **ninguna manera** se debe usar busy waiting ni esperas arbitrarias.

Ej. 9 ("Hello" tolerante a fallas). El siguiente programa crea un proceso que imprime "Hello" a intervalos regulares. Por una falla desconocida termina al poco tiempo con un error.

```
-module(hello).
-export([init/0]).

hello() ->
        receive after 1000 -> ok end,
        io:fwrite("Hello ~p~n", [case rand:uniform(10) of 10 -> 1/uno; _ -> self() end]),
        hello().

init() -> spawn(fun () -> hello() end).
```

Reemplace este proceso por dos, donde el segundo deba levantar al proceso que imprime "Hello" cada vez que se caiga.

Nota: puede ser de ayuda utilizar `process_flag(trap_exit, true)`.

Ej. 10 (Cambio en Caliente). El cliente está satisfecho con el servicio de salutación, pero le gustaría que lo salude en castellano y no en inglés. Modifique el código de manera que, una vez levantado el servicio, se pueda cambiar el mensaje por "Hola" sin darlo de baja. Es decir, que se pueda hacer lo siguiente:

```
2> hello:init().
...
Hello <0.XX.0>
Hello <0.XX.0>
```

Después reemplazar `Hello` por `Hola` en `hello.erl`, y

```
3> c(hello).
Hello <0.XX.0>
Hola <0.XX.0>
...
```

Notar que el PID del proceso no debería cambiar si el proceso no muere por el error.