Lenguajes de programación

Programación Concurrente en Erlang

Procesos y Concurrencia en Erlang

- En Erlang, la Concurrencia se implementa mediante la **creación y comunicación de procesos**.
- **Proceso**: unidad de cómputo separada y auto-contenida que se ejecuta concurrentemente con otros procesos en el sistema.
- Los procesos de Erlang **no comparten memoria (datos)** con otros procesos.
- Los procesos se comunican mediante el paso de mensajes (**modelo de programación concurrente**)

Procesos

- En Erlang, los procesos pertenecen al lenguaje de programación y no al Sistema Operativo.
- En Erlang, la programación con procesos es fácil, ya que solo necesitas 3 primitivos:
  - **spawn:** para crear procesos,
  - **send**: para mandar mensajes y
  - **receive:** para recibir mensajes.

Creación de Procesos

- **spawn/1** o **spawn/3** crean un nuevo proceso concurrente y regresa su identificador.

```
Pid = spawn(Función)
Pid = spawn(Módulo, Función, ListaArgs)
```

- Los identificadores de procesos (**pid**) son usados para intercambio de mensajes
- La llamada **no espera** a que la función se evalúe (regresa inmediatamente).
- El proceso **termina automáticamente** cuando la función que lo creó se completa.
- El **valor de retorno** del proceso se pierde

Ejemplo

Escribe su argumento N veces

Crea 2 procesos concurrentes **di_algo**

```
-module(habla).
-export([inicio/0, di_algo/2]).

di_algo(_, 0) ->
        hecho;
di_algo(Que, Veces) ->
        io:format("~p~n", [Que]),
        di_algo(Que, Veces - 1).

inicio() ->
        spawn(habla, di_algo, [hola, 3]),
        spawn(habla, di_algo, [adios, 3]).
```

Ejemplo

Identificador del 2do. proceso (último)

```
5> c(habla).
{ok,habla}

6> habla:di_algo(hola,3).
hola
hola
hola
hecho

7> habla:inicio().
hola
adios
hola
adios
<0.44.0>
hola
adios
```

Envío de Mensajes

- Un mensaje se envía a otro proceso mediante el primitivo '!' (***send***), como:

```
Pid ! Mensaje
```

- `Pid` es el identificador del proceso al que se le envía el mensaje.
- El envío del mensaje es **asíncrono**
  - El que envía el mensaje continúa con lo que estaba haciendo (**no espera**)
  - El sistema no informa al que envía si el mensaje se entregó, aún ni cuando el proceso destino ya no existiera
  - La aplicación debe implementar todas las formas de chequeo requerido

Envío de Mensajes

- El **mensaje** puede ser cualquier término Erlang válido
- El **valor de retorno de !** es el mensaje que envía, de forma que:

```
Pid1 ! Pid2 ! … ! Mensaje
```

- Le enviaría el mismo mensaje a todos los procesos `Pid1`, `Pid2`, …
- Si el receptor no ha terminado, todos los mensajes le son entregados en el mismo orden en el que se le envían

Recepción de Mensajes

- El primitivo **receive** es usado para recibir mensajes, con la siguiente sintaxis:

```
receive
    Patrón1 [when Guardia1] ->
          Acciones1;
    Patrón2 [when Guardia2] ->
          Acciones2;
    ...
end
```

- Cada proceso tiene su propio **buzón de correo**
- Todos los mensajes que se le envían a un proceso se almacenan en su buzón en el orden en el que se reciben

Ejemplo

```
-module(servidor_area).
-export([ciclo/0]).

ciclo() ->
  receive
    {rectangulo, Anchura, Altura} ->
        io:format("Area del rectangulo = ~p~n" ,[Anchura * Altura]),
        ciclo();
    {circulo, R} ->
        io:format("Area del circulo es ~p~n" , [3.14159 * R * R]),
        ciclo();
    Otro ->
        io:format("Desconozco el area del ~p ~n" ,[Otro]),
        ciclo()
  end.
```

Ejemplo

```
1> Pid = spawn(fun servidor_area:ciclo/0).
<0.36.0>
2> Pid ! {rectangulo,6,10}.
Area del rectangulo = 60
{rectangulo,6,10}
3> Pid ! {circulo, 23}.
Area del circulo = 1661.90
{circulo,23}
4> Pid ! {triangulo,2,4,5}.
Desconozco el area del {triangulo,2,4,5}
{triangulo,2,4,5}
```

Recepción de Mensajes

- Cuando un mensaje es recibido, el sistema trata de **empatarlo secuencialmente** con alguno de los patrones (y con sus posibles guardias).
  - Si un **mensaje empata** con algún patrón, se elimina del buzón y se evalúan sus acciones relacionadas
  - `receive` regresa el valor de la última expresión evaluada en las acciones
  - Si un **mensaje no empata** con ningún patrón, permanece en el buzón para su procesamiento posterior y el proceso continua con el siguiente mensaje en su buzón

Recepción de Mensajes

- El proceso que evalúa un **`receive`** es **suspendido** hasta que un mensaje sea empatado
- Los mensajes que arriban a un proceso **no pueden bloquear** otros mensajes para ese proceso
- El **buzón se puede llenar** con mensajes que no empatan con los patrones
- Es **responsabilidad del programador** asegurarse que el buzón no se llene

Mensajes de Procesos Específicos

- Cuando se quiere recibir mensajes de un proceso específico el que envía debe mandar su propio **pid** en el mensaje

```
Pid ! {self(),abc}
```

- La función **self/0** regresa su pid al proceso que la llama
- Este mensaje puede ser recibido por

```
receive
   {Pid, Msg} ->
       ...
end
```

- Permitiendo recibir mensajes solo de este proceso.

OTRO EJEMPLO

```
-module(pingpong).
-export([inicio/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
        Pong_PID ! terminado,
        io:format("Ping termino~n", []);
ping(N, Pong_PID) ->
        Pong_PID ! {ping, self()},
        receive
                pong -> io:format("Ping recibe pong~n", [])
        end,
        ping(N - 1, Pong_PID).

pong() ->
        receive
                terminado -> io:format("Pong termino~n", []);
                {ping, Ping_PID} ->
                        io:format("Pong recibio ping~n", []),
                        Ping_PID ! pong,
                        pong()
        end.

inicio() ->
        Pong_PID = spawn(pingpong, pong, []),
         spawn(pingpong, ping, [3, Pong_PID]).
```

Otro Ejemplo

```
2> pingpong:inicio().
Pong recibio ping
Ping recibio pong
<0.36.0>
Pong recibio ping
Ping recibio pong
Pong recibio ping
Ping recibio pong
ping termino
Pong termino
```

Tiempos de Espera

- El primitivo receive puede incluir **tiempos de espera** (*Timeouts*) para no bloquear al proceso para siempre si no recibe un mensaje

- **Sintaxis:**

```
receive
    Mensaje1 [when Guardia1] ->
          Acciones1 ;
    Mensaje2 [when Guardia2] ->
          Acciones2 ;
    ...
after
    EsperaExpr ->
          AccionesEspera
end
```

Tiempos de Espera

- **`EsperaExpr`** se evalúa a un entero interpretado como un tiempo en **milisegundos**
- Las **`AccionesEspera`** se evalúan si no se selecciona un mensaje antes que termine el tiempo de espera
- **Ejemplo**: suspender un proceso T milisegundos

```
duerme(T) ->
    receive
    after T ->
          true
    end.
```

Otro Ejemplo: Detectar Dobles Clicks

```
get_event() ->
      receive
            {mouse, click} ->
                  receive
                        {mouse, click} ->
                              double_click
                  after
                        double_click_interval() ->
                              single_click
                  end
            ...
      end.
```

Tiempos Especiales de Espera

- Hay 2 tiempos especiales de espera:
  - **infinity**: especifica una espera que nunca ocurrirá.
    - Útil si el tiempo de espera se calcula en tiempo real (fuera del **`receive`**)
  - **0**: especifica que la espera se acaba inmediatamente
    - Pero antes el sistema trata todos los mensajes actualmente en el buzón

Ejemplo de Espera 0

- Para eliminar todos los mensajes del buzón de entrada de un proceso

```
borra_mensajes() ->
      receive
            _Any ->
                  flush_buffer()
      after 0 ->
            true
      end.
```

- Sin la espera 0 se bloquearía hasta que hubiera algún mensaje que eliminar

Otro Ejemplo de Espera 0

- Implementa una forma de recepción con prioridades

```
recepcion_prioritaria() ->
      receive
            {urgente, X} ->
                  {urgente, X}
      after 0 ->
            receive
                  Cualquiera ->
                        Cualquiera
            end
      end
```

Registro de Procesos

- El PID de un proceso se requiere para mandarle un mensaje
  - Esto es **muy seguro**, pero **inconveniente** porque el proceso le tiene que enviar su PID a todos los otros procesos que se quieran comunicar con él
- Erlang tiene un **método para publicar los PIDs** para que cualquier proceso en el sistema les pueda enviar mensajes
- El método se conoce como **registro de procesos**

Registro de Procesos

- **Primitivos Predefinidos:**
  - **`register(Alias, Pid)`** – registra el proceso **`Pid`** con el nombre **`Alias`** (un átomo)
  - **`unregister(Alias)`** – remueve cualquier registro con el nombre **`Alias`**
  - **`whereis(Alias)-> Pid | undefined`** – determina si el nombre **`Alias`** está registrado
  - **`registered()`** – regresa una lista con todos los procesos registrados en el sistema

Ejemplo de Registro de Procesos

```
-module(reloj).
-export([inicia/2, para/0]).
inicia(Tiempo, Funcion) ->
  register(reloj, spawn(fun() ->
      tictac(Tiempo, Funcion) end)).
para() -> reloj ! para.
tictac(Tiempo, Funcion) ->
      receive
            para -> void
      after Tiempo ->
            Funcion(),
            tictac(Tiempo, Funcion)
      end.
```

Ejemplo de Registro de Procesos

- Para hacer que el reloj haga tictac y despliegue la marca de tiempo cada 5 segundos:

```
3> reloj:inicia(5000, fun() ->
io:format("TICTAC ~p~n",[erlang:now()])
end).
true
TICTAC {1414,771148,424000}
TICTAC {1414,771153,432000}
TICTAC {1414,771158,439000}
TICTAC {1414,771163,447000}
4> reloj:para().
para
```