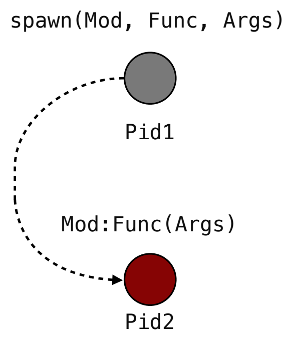
Erlang Solutions Ltd.

Concurrent Erlang

Overview: concurrent Erlang I

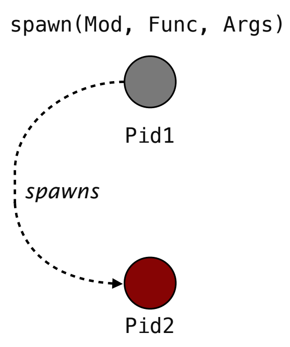
- Concurrent Erlang I
  - Creating Processes
  - Message Passing
  - Receiving Messages
  - Data in Messages
- Concurrent Erlang II

Creating Processes



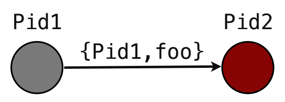
- Before
  - Code executed by Process 1
  - **process identifier** is Pid1
  - `Pid2 = spawn(M, F, A)`
- After
  - A new process with Pid2 is created
  - Pid2 is only known to Pid1
  - Pid2 runs M:F(A)
  - M:F/Arity must be exported
- Convention: we identify processes by their process ids (pids)

Creating Processes



- The BIF **spawn** never fails
- A process terminates
  - **abnormally** when run-time errors occur
  - **normally** when there is no more code to execute

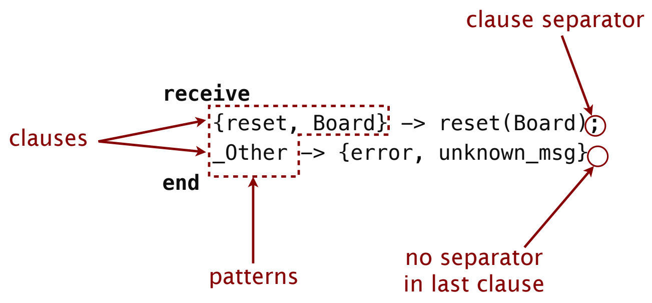
Message Passing



```
Pid2 ! {self(),foo}
```

- Messages are sent using the **Pid ! Msg** expression
  - **Msg** is any valid Erlang data type
- Sending a message will never fail
- Messages sent to non-existing processes are thrown away
- Received messages are stored in the process' mailbox

Message Passing



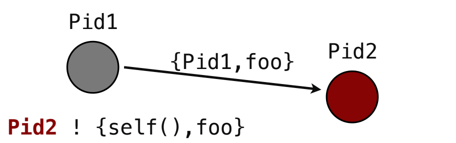
```
receive
  {reset, Board} -> reset(Board);
  _Other -> {error, unknown_msg}
end
```

Receiving Messages

```
receive
    Pattern1 ->
         <expression 1>,
         <expression 2>,
         ...,
         <expression N>;
    Pattern2 ->
         <expression 1>,
         ...,
         <expression N>;
    ...;
    PatternN ->
         <expression 1>,
         ...
         <expression N>
end
```

- Messages are retrieved using a **receive** clause
- **receive** suspends the process until a message is received
- Message passing is asynchronous

Receiving Messages

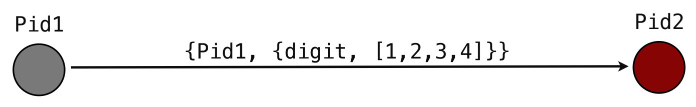


```
Pid2 ! {self(),foo}
                        receive
                          start -> ...
                          stop  -> ...
                          {Pid, foo} ->
                            ...
                        end
```

- Messages can be matched and selectively retrieved
- Messages are received when a message matches a clause
- Mailboxes are scanned sequentially.

Receiving Messages

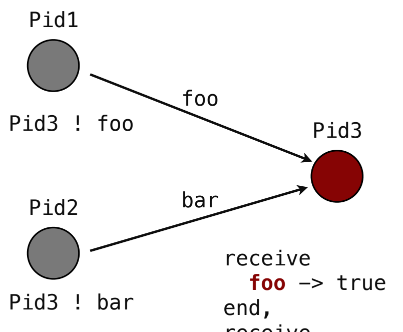
```
Pid2 ! {self(), {digit, [1,2,3,4]}}
```



```
receive
  {Pid,{digit,Digit}} ->
     ...
end
```

- If **Pid** is bound before receiving the message, then only data tagged with that pid can be pattern matched
- The variable **Digit** is bound when receiving the message

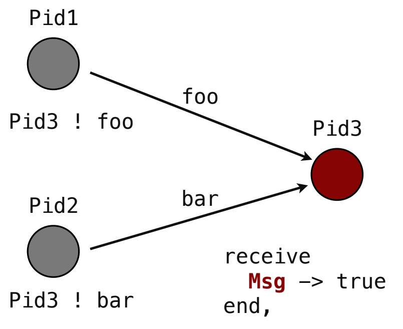
Receiving Messages: selective



```
receive
  foo -> true
end,
receive
  bar -> true
end,
...
```

- The message **foo** is received, followed by the message **bar**
- This is irrespective of the order in which they were sent or stored in the mailbox

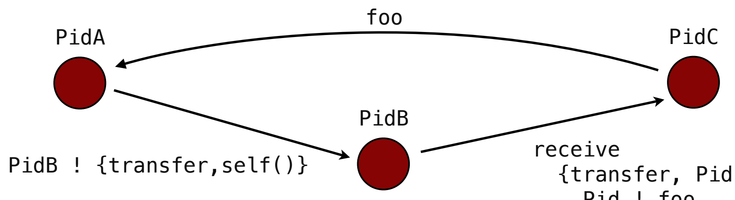
Receiving Messages: non-selective



```
receive
  Msg -> true
end,
...
```

- The first message to arrive at the process **Pid3** will be processed
- The variable **Msg** in the process **Pid3** will be bound to one of the atoms **foo** or **bar** depending on which arrives first.

Receiving Messages



```
PidB ! {transfer,self()}
```

```
receive
  {transfer, Pid} ->
    PidC ! {transfer, Pid}
end
```

```
receive
  {transfer, Pid} ->
    Pid ! foo
end
```

- PidA sends a message to PidB containing its own Pid
- PidB binds it to variable A and sends a message to PidC
- PidC receives the message and replies directly to PidA

Data in Messages: example

```
-module(echo).
-export([go/0, loop/0]).

go() ->
  Pid= spawn(echo,loop,[]),
  Pid ! {self(), hello},
  receive
    {Pid, Msg} ->
    io:format("~w~n",[Msg])
  end,
  Pid ! stop.
```

```
loop() ->
  receive
    {From, Msg} ->
      From ! {self(), Msg},
      loop();
    stop ->
      true
  end.
```

Summary: concurrent Erlang I

- Concurrent Erlang I
  - Creating Processes
  - Message Passing
  - Receiving Messages
  - Data in Messages
- Concurrent Erlang II

Overview: concurrent Erlang II

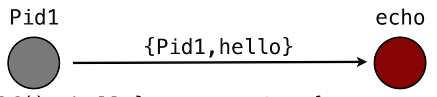
- Concurrent Erlang I
- Concurrent Erlang II
  - Registered Processes
  - Timeouts
  - More on Processes
  - The Process Manager

Registered Processes

```
register(Alias, Pid)
  Alias ! Message
```

- Registers the process **Pid** with the name **Alias**
- Any process can send a message to a registered process
- The BIF **registered/0** returns all registered process names
- The BIF **whereis(Alias)** returns the Pid of the process with the name **Alias**.

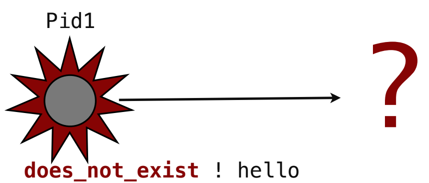
Registered Processes



```
echo ! {self(), hello}      receive {From, Msg} -> ... end
```

```
go() -> register(echo, spawn(echo, loop, [])).

loop() ->
   receive
     {From, Msg} ->
       From ! {self(), Msg},
       loop();
     stop -> true
   end.
```

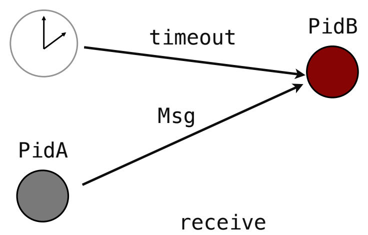
Message Passing



`does_not_exist ! hello`

- Sending messages to non-existing registered processes causes the calling process to terminate with a **badarg** error

Timeouts



```
receive
  Msg ->
    <expressions1>
  after TimeOut ->
    <expressions2>
end
```

- If the message **Msg** is received within the time **TimeOut**, <expressions1> will be executed
- If not, <expressions2> will be executed
- TimeOut is an integer denoting the time in milliseconds or the atom **infinity**

Timeouts

```
read(Key) ->
    flush(),
    db ! {self(), {read, Key}},
    receive
      {read, R} ->
         {ok, R};
      {error, Reason} ->
         {error, Reason}
    after 1000 ->
      {error, timeout}
    end.
```

- If the server takes more than a second to handle the request, a timeout is generated
- Do not forget to handle messages received after a timeout

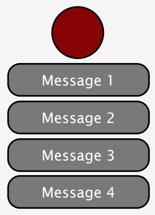
Timeouts

```
send_after(Time, Msg) ->
    spawn(timer,
          send,
          [self(),Time,Msg]).

send(Pid, Time, Msg) ->
    receive
    after Time ->
        Pid ! Msg
    end.

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

- **send_after(T, What)** sends the message **What** to the current process after **T** milliseconds
- The **sleep(T)** function will suspend the calling process for **T** milliseconds

Timeouts

```
flush() ->
   receive
      _ -> flush()
   after 0 ->
      ok
   end.
```



- **flush()** will clear the mailbox from all messages, stopping when it is empty.

More on Processes: **definitions**

Process
A concurrent activity. The system may have many concurrent processes executing at the same time

Message
A method of communication and sharing data between processes

Timeout
A mechanism for waiting for a given period of time for an incoming message

More on Processes: **definitions**

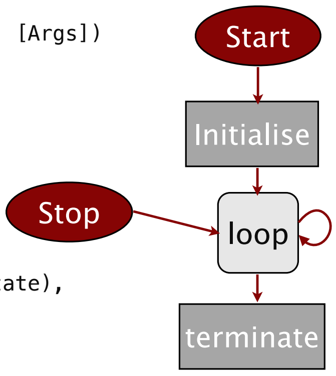
Registered Processes
Processes which have been given a name with BIFs such as **register/2**.

Termination
A process is said to terminate normally when it has no more code to execute.

It terminates abnormally if a run time error occurs or if someone makes it exit with a non-normal reason.

More on Processes: process skeleton

```
start(Args) -> spawn(server, init, [Args])

init(Args) ->
  State = initialize_state(Args),
  loop(State).

loop(State) ->
   receive
      {handle, Msg} ->
         NewState = handle(Msg, State),
         loop(NewState);
      stop -> terminate(State)
   end.

terminate(State) -> clean_up(State).
```

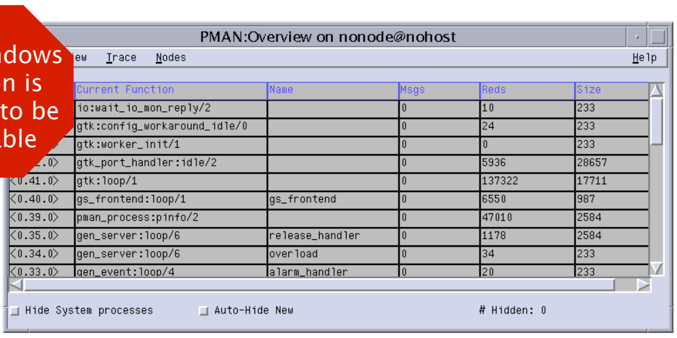


The Process Manager

- Used to inspect the state of processes in a local or distributed Erlang system
- Trace output for messages sent & received
- Trace output for process events such as spawn, exit and link
- Trace output for BIF and function calls

The Process Manager

The windows version is known to be unstable

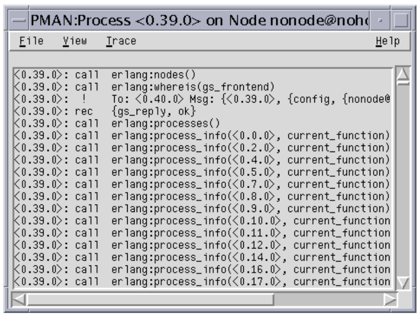


	Current Function	Name	Msgs	Reds	Size
[illegible]	io:wait_io_mon_reply/2		0	10	233
[illegible]	gtk:config_workaround_idle/0		0	24	233
[illegible]	gtk:worker_init/1		0	0	233
[illegible]	gtk_port_handler:idle/2		0	5936	28657
<0.41.0>	gtk:loop/1		0	137322	17711
<0.40.0>	gs_frontend:loop/1	gs_frontend	0	6550	987
<0.39.0>	pman_process:pinfo/2		0	47010	2584
<0.35.0>	gen_server:loop/6	release_handler	0	1178	2584
<0.34.0>	gen_server:loop/6	overload	0	34	233
<0.33.0>	gen_event:loop/4	alarm_handler	0	20	233

☐ Hide System processes ☐ Auto-Hide New # Hidden: 0

- pman:start()

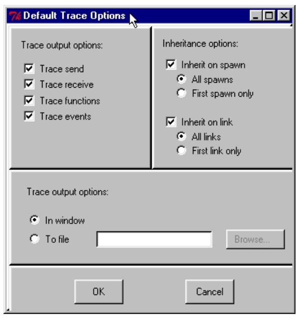
Process Manager: processes



```
<0.39.0>: call  erlang:nodes()
<0.39.0>: call  erlang:whereis(gs_frontend)
<0.39.0>:  !    To: <0.40.0> Msg: {<0.39.0>, {config, {nonode@
<0.39.0>: rec   {gs_reply, ok}
<0.39.0>: call  erlang:processes()
<0.39.0>: call  erlang:process_info(<0.0.0>, current_function)
<0.39.0>: call  erlang:process_info(<0.2.0>, current_function)
<0.39.0>: call  erlang:process_info(<0.4.0>, current_function)
<0.39.0>: call  erlang:process_info(<0.5.0>, current_function)
<0.39.0>: call  erlang:process_info(<0.7.0>, current_function)
<0.39.0>: call  erlang:process_info(<0.8.0>, current_function)
<0.39.0>: call  erlang:process_info(<0.9.0>, current_function)
<0.39.0>: call  erlang:process_info(<0.10.0>, current_function
<0.39.0>: call  erlang:process_info(<0.11.0>, current_function
<0.39.0>: call  erlang:process_info(<0.12.0>, current_function
<0.39.0>: call  erlang:process_info(<0.14.0>, current_function
<0.39.0>: call  erlang:process_info(<0.16.0>, current_function
<0.39.0>: call  erlang:process_info(<0.17.0>, current_function
```

- Prints the trace messages and process state

Process Manager: options



Trace output options:
- [✓] Trace send
- [✓] Trace receive
- [✓] Trace functions
- [✓] Trace events

Inheritance options:
- [✓] Inherit on spawn
  - [✓ All spawns]
  - First spawn only
- [✓] Inherit on link
  - [✓ All links]
  - First link only

Trace output options:
- [✓ In window]
- To file

- Pick what trace messages you want to view
- Pick the inheritance level when spawning

Summary: concurrent Erlang

- Concurrent Erlang I
  - Creating Processes
  - Message Passing
  - Receiving Messages
  - Data in Messages
- Concurrent Erlang II
  - Registered Processes
  - Timeouts
  - More on Processes
  - The Process Manager