www.erlang-solutions.com

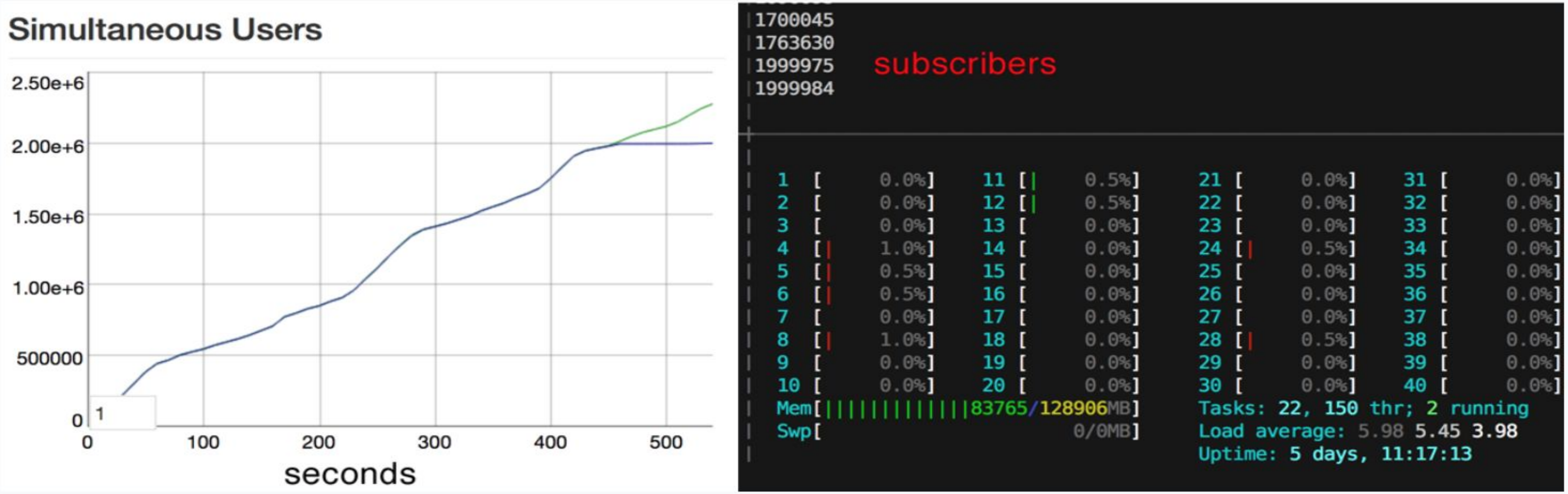
Concurrent Erlang

Overview: concurrent Erlang

- Creating Processes
- Message Passing
- Receiving Messages
- Data in Messages
- Registered Processes
- Timeouts
- More on Processes
- Observer Processes

The Road to 2 Million Websocket Connections in Phoenix

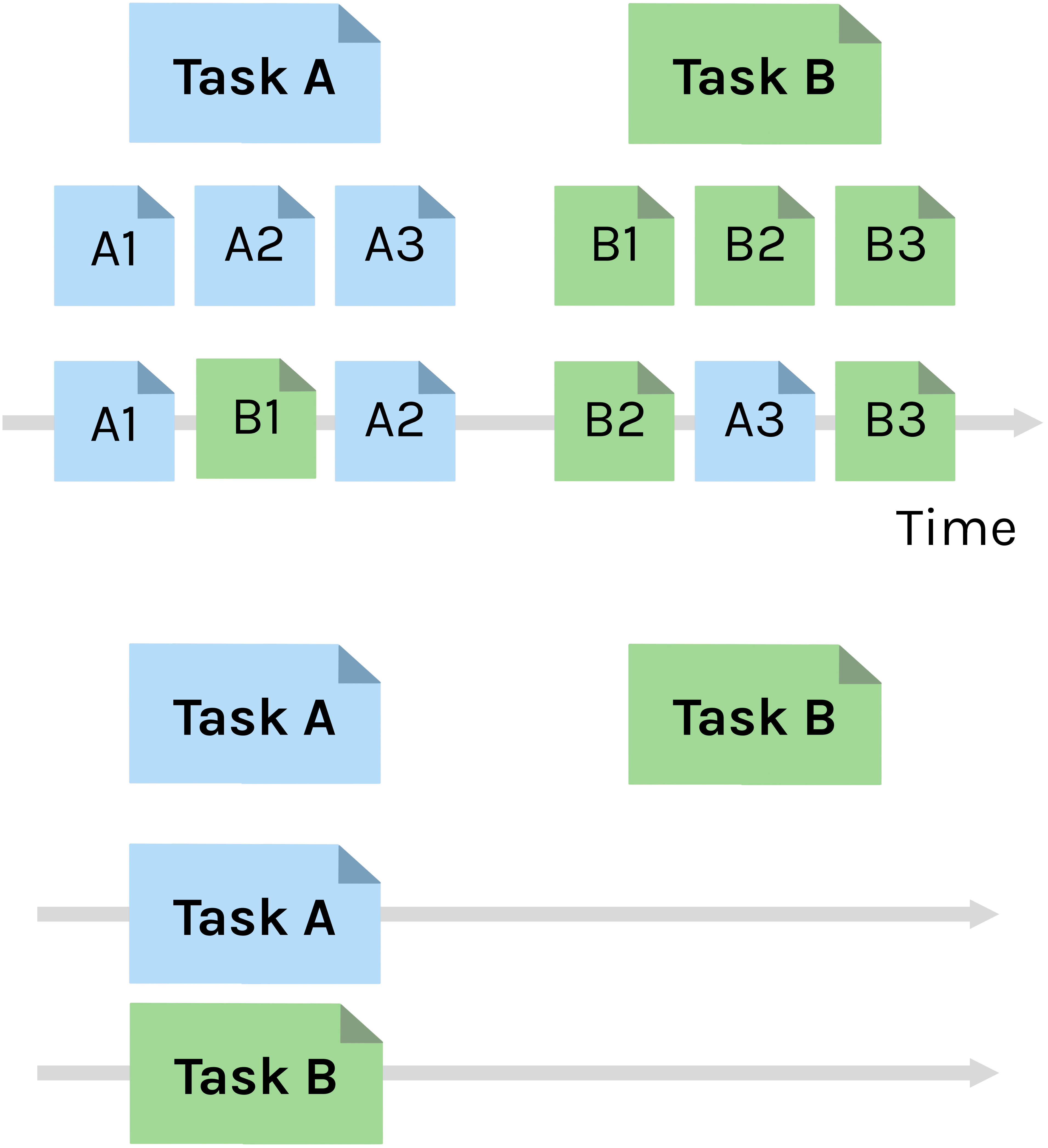
Posted on November 3rd, 2015 by Gary Rennie



```
1700045
1763630
1999975
1999984

  1  [        0.0%]   11 [|       0.5%]   21 [        0.0%]   31 [        0.0%]
  2  [        0.0%]   12 [|       0.5%]   22 [        0.0%]   32 [        0.0%]
  3  [        0.0%]   13 [        0.0%]   23 [        0.0%]   33 [        0.0%]
  4  [|       1.0%]   14 [        0.0%]   24 [|       0.5%]   34 [        0.0%]
  5  [|       0.5%]   15 [        0.0%]   25 [        0.0%]   35 [        0.0%]
  6  [|       0.5%]   16 [        0.0%]   26 [        0.0%]   36 [        0.0%]
  7  [        0.0%]   17 [        0.0%]   27 [        0.0%]   37 [        0.0%]
  8  [|       1.0%]   18 [        0.0%]   28 [|       0.5%]   38 [        0.0%]
  9  [        0.0%]   19 [        0.0%]   29 [        0.0%]   39 [        0.0%]
  10 [        0.0%]   20 [        0.0%]   30 [        0.0%]   40 [        0.0%]
  Mem[||||||||||||83765/128906MB]         Tasks: 22, 150 thr; 2 running
  Swp[                     0/0MB]         Load average: 5.98 5.45 3.98
                                          Uptime: 5 days, 11:17:13
```

If you have been paying attention on Twitter recently, you have likely seen some increasing numbers regarding the number of simultaneous connections the Phoenix web framework can handle. This post documents some of the techniques used to perform the benchmarks.

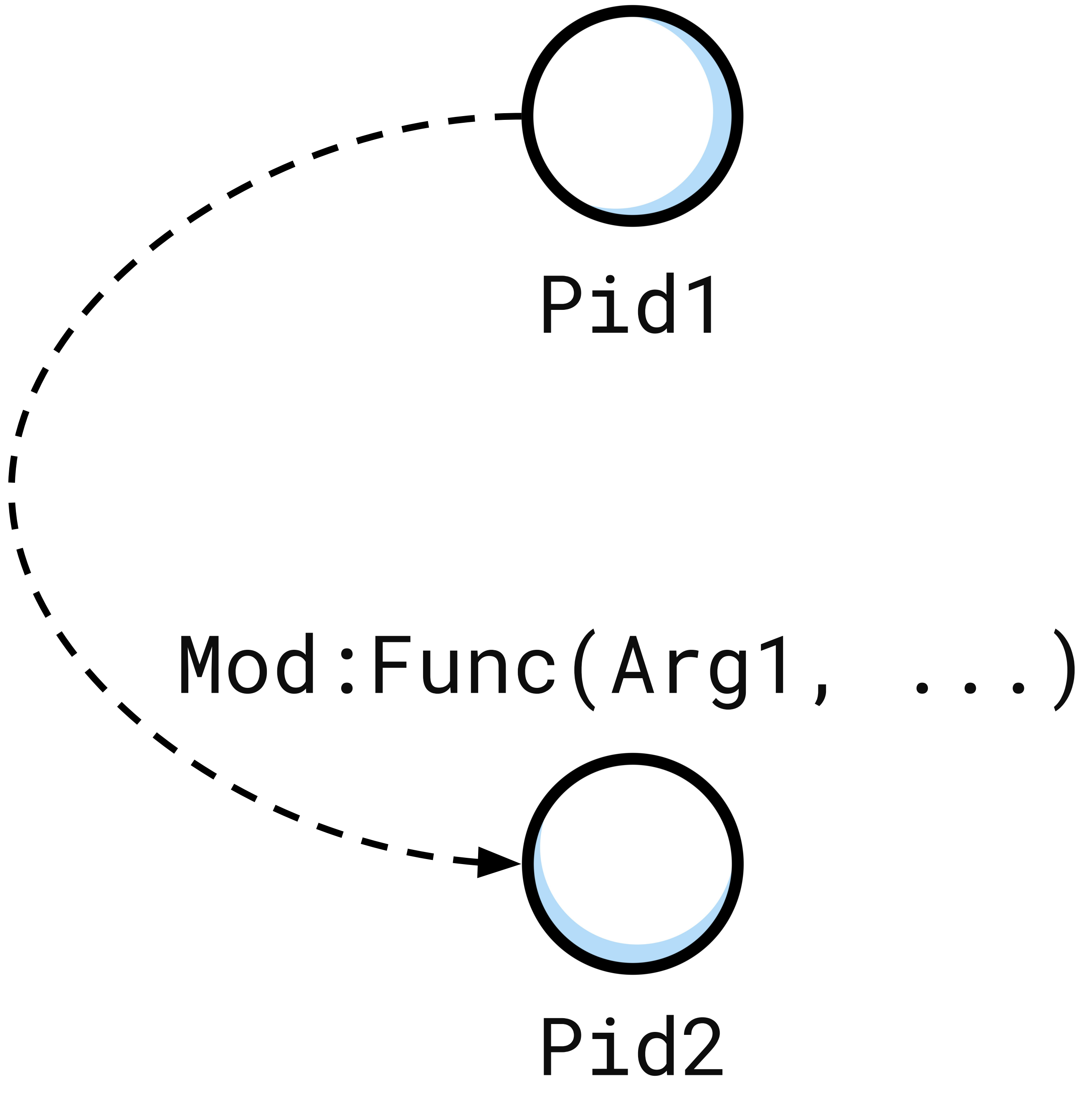
Concurrency vs Parallelism



- ▶ Concurrency happens when your code is running in different processes
- ▶ Control of Concurrency is key to scale
- ▶ Concurrent solutions can exploit the underlying system's parallelism, if present
- ▶ Parallelism can speed up execution

Creating Processes

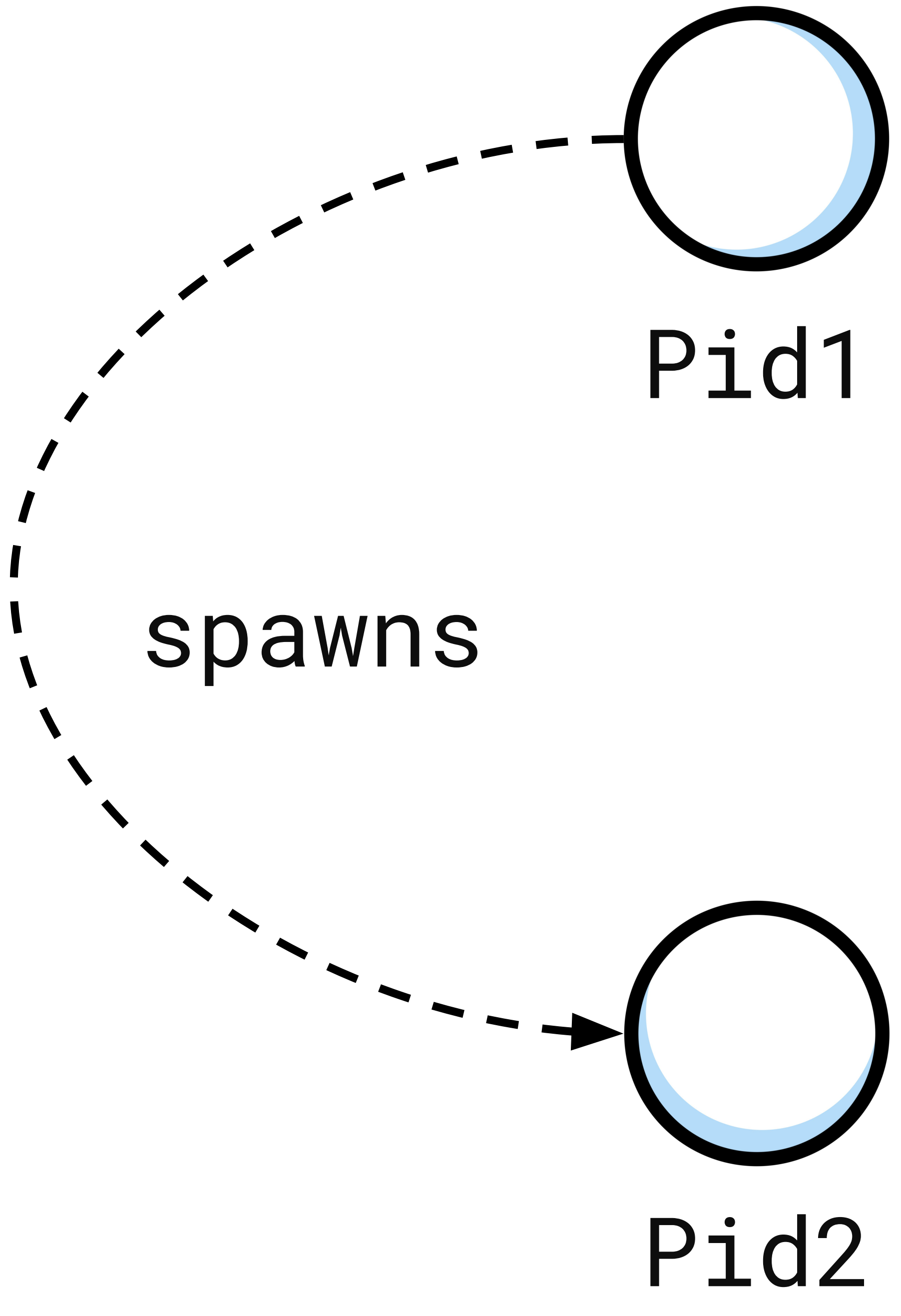
```
Pid2 = spawn(Mod, Func, Args)
```



- 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)

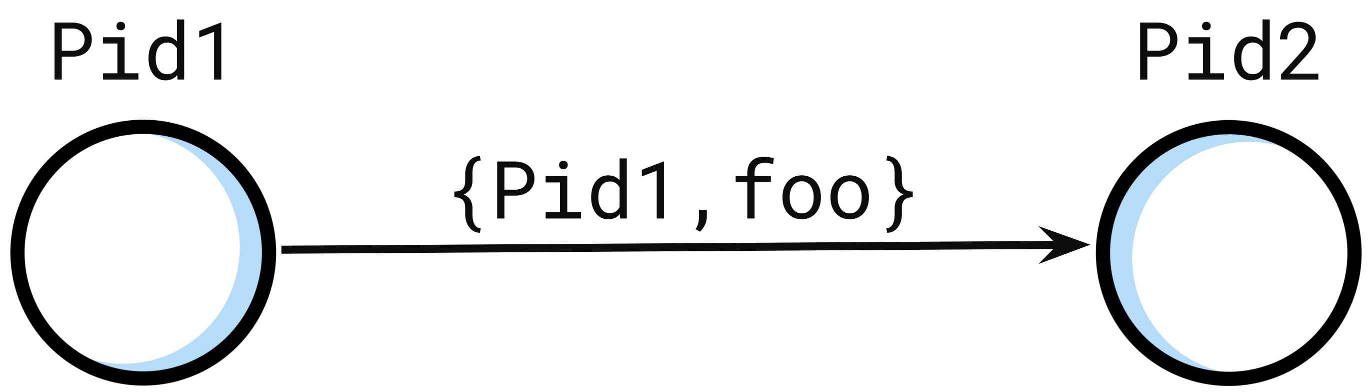
Creating Processes

```
Pid2 = spawn(Mod, Func, Args)
```



- ▶ The BIF **spawn** fails when max number of running processes reached
- ▶ A process terminates
  - ○ **abnormally** when run-time errors occur
  - ○ **normally** when there is no more code to execute

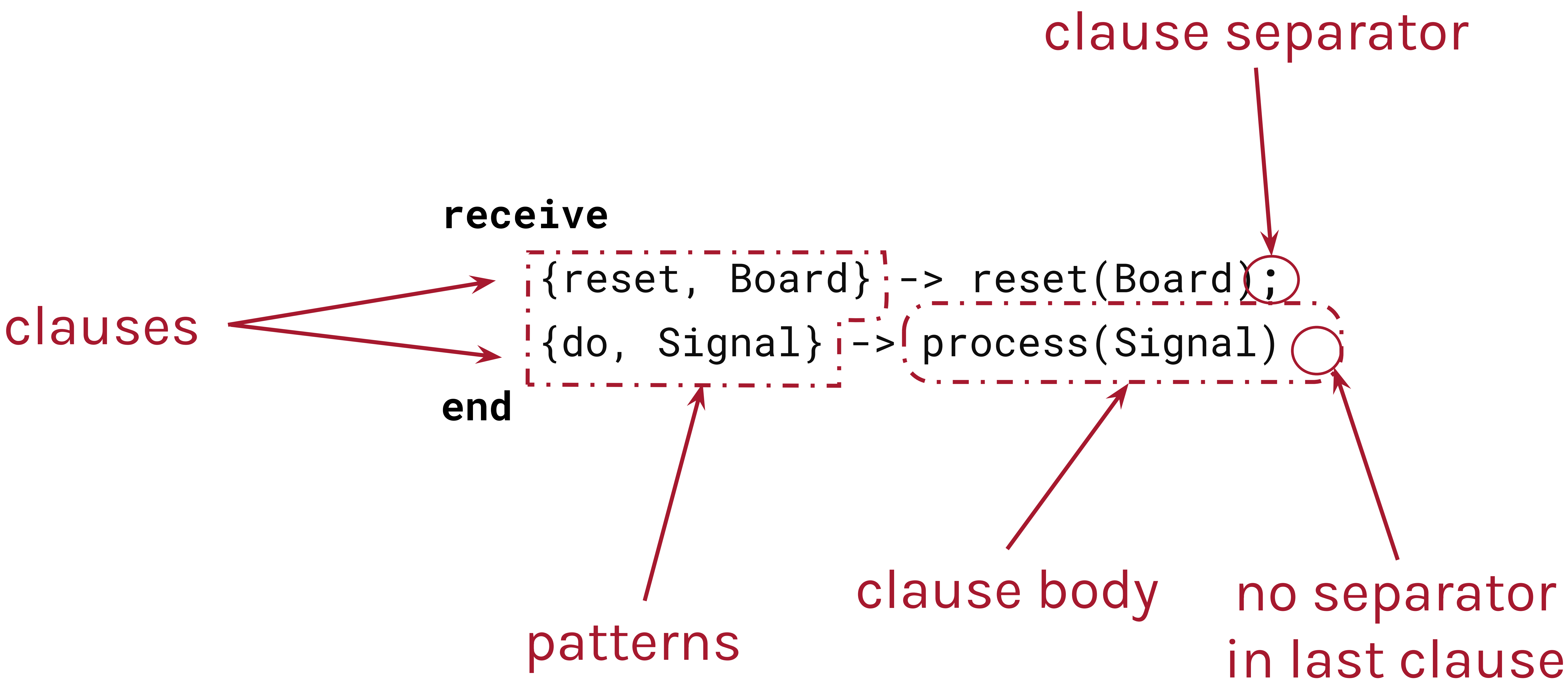
Message Passing



```
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);
    {do, Signal} -> process(Signal)
end
```

clause separator

clauses

patterns

clause body

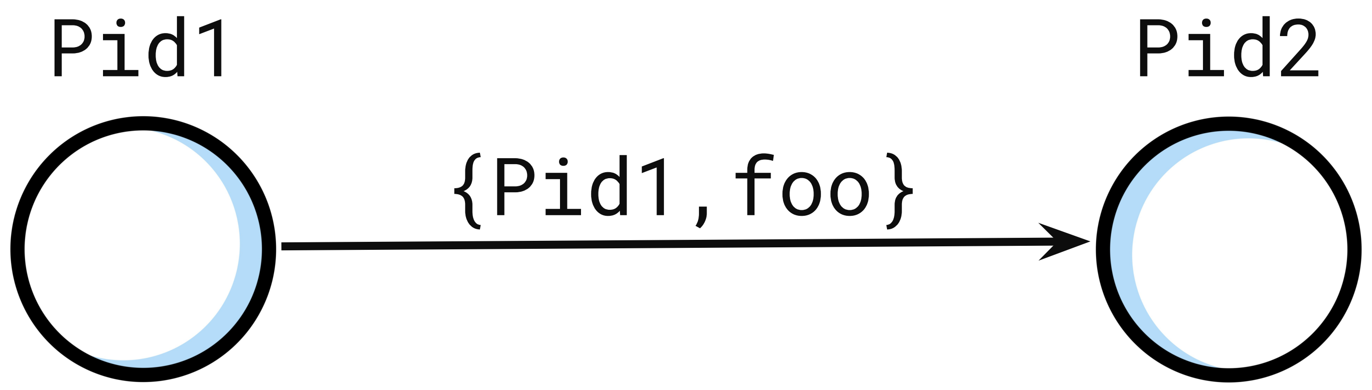
no separator
in last clause

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 matching message is received
- Message passing is asynchronous

Receiving Messages



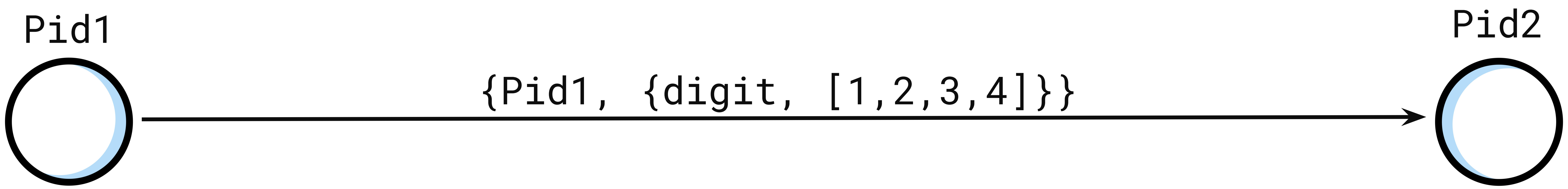
```
Pid2 ! {self(),foo}
```

```
receive
  start      -> start_it();
  stop       -> stop_it();
  {Pid, foo} -> foo(Pid)
end
```

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

Receiving Messages

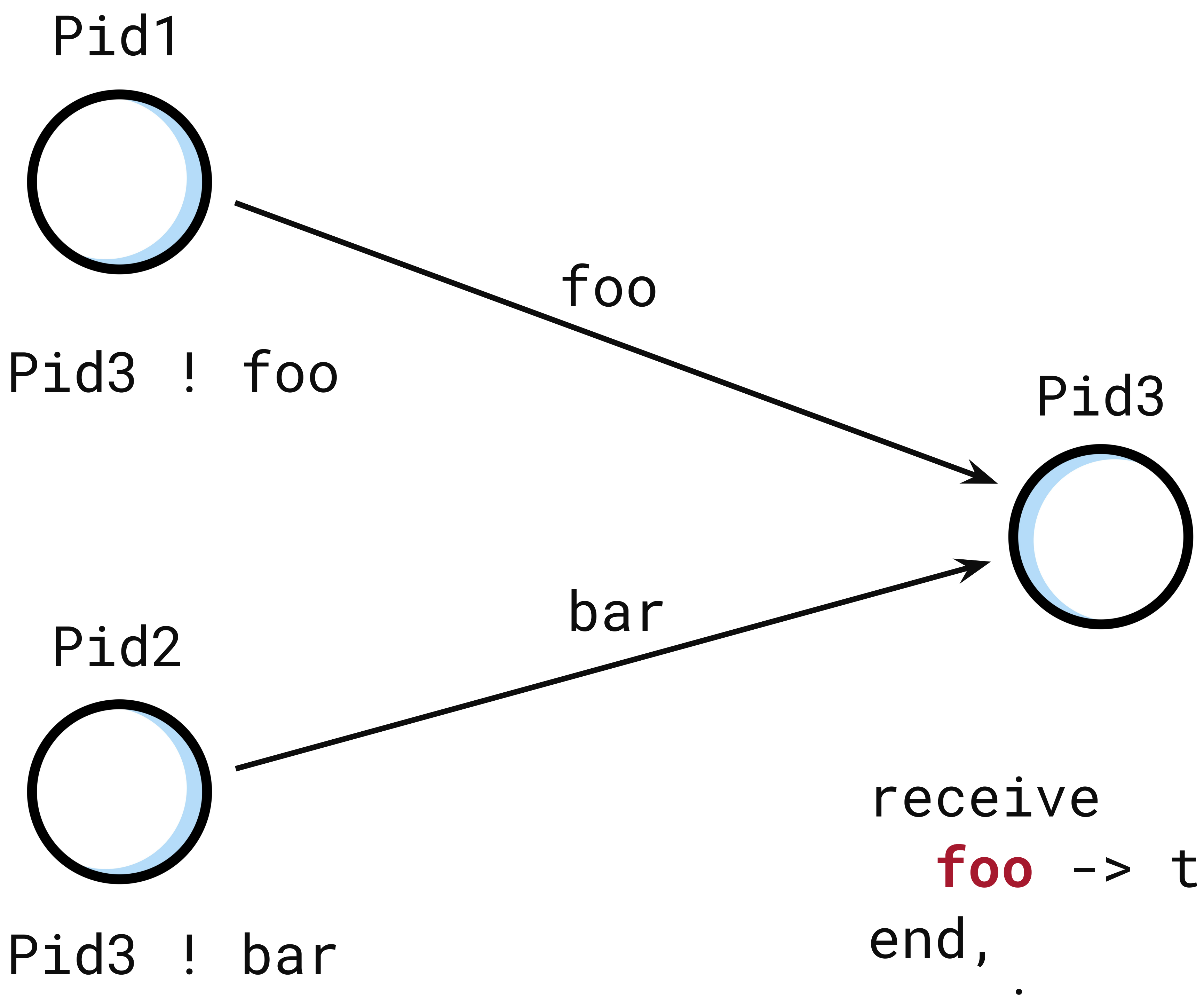
```
Pid2 ! {self(), {digit, [1,2,3,4]}}
```



```
receive
  {Pid,{digit,Digit}} ->
     digit(Pid, 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

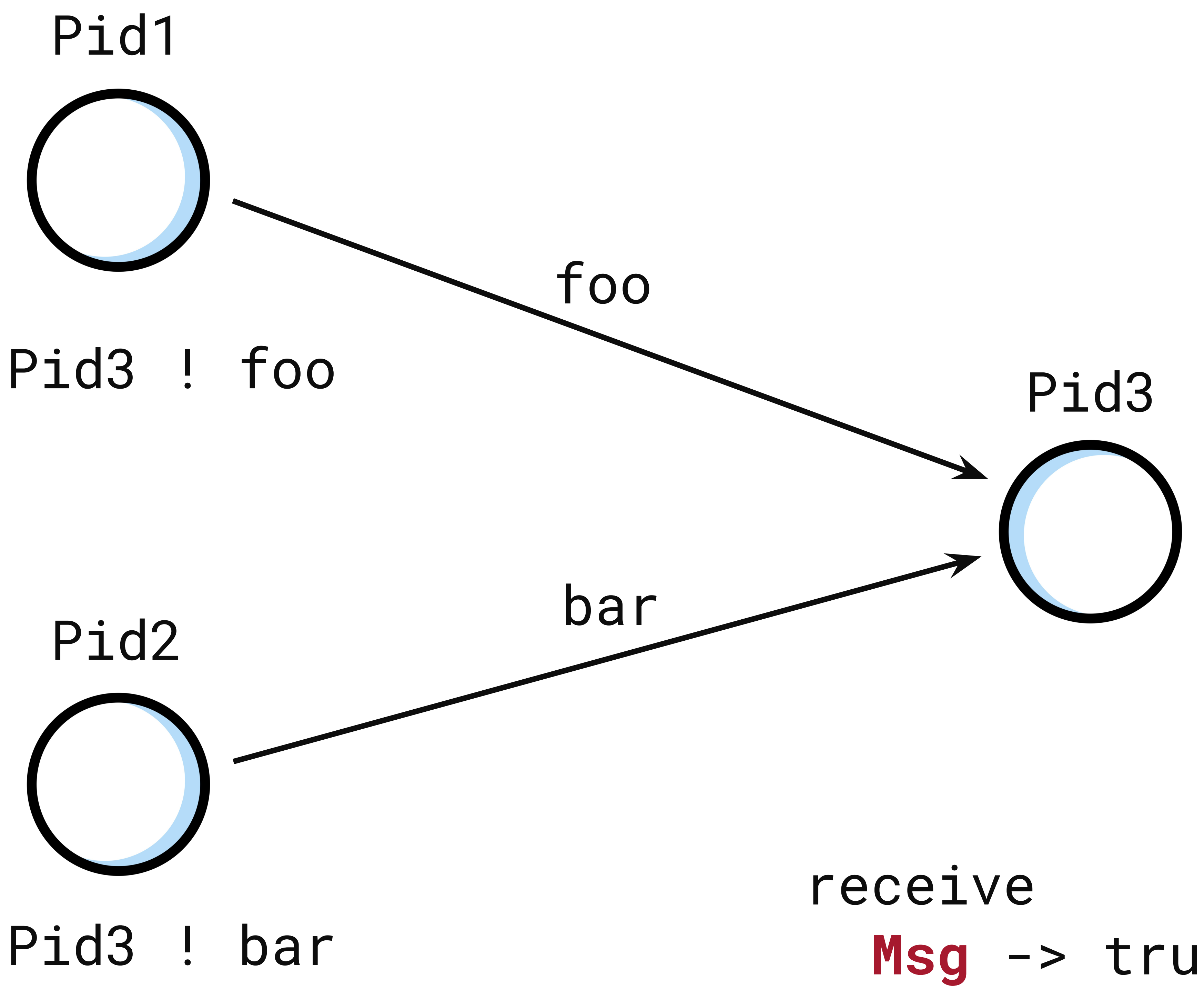
Receiving Messages: selective



```
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


Pid1

Pid3 ! foo

foo

Pid3

bar

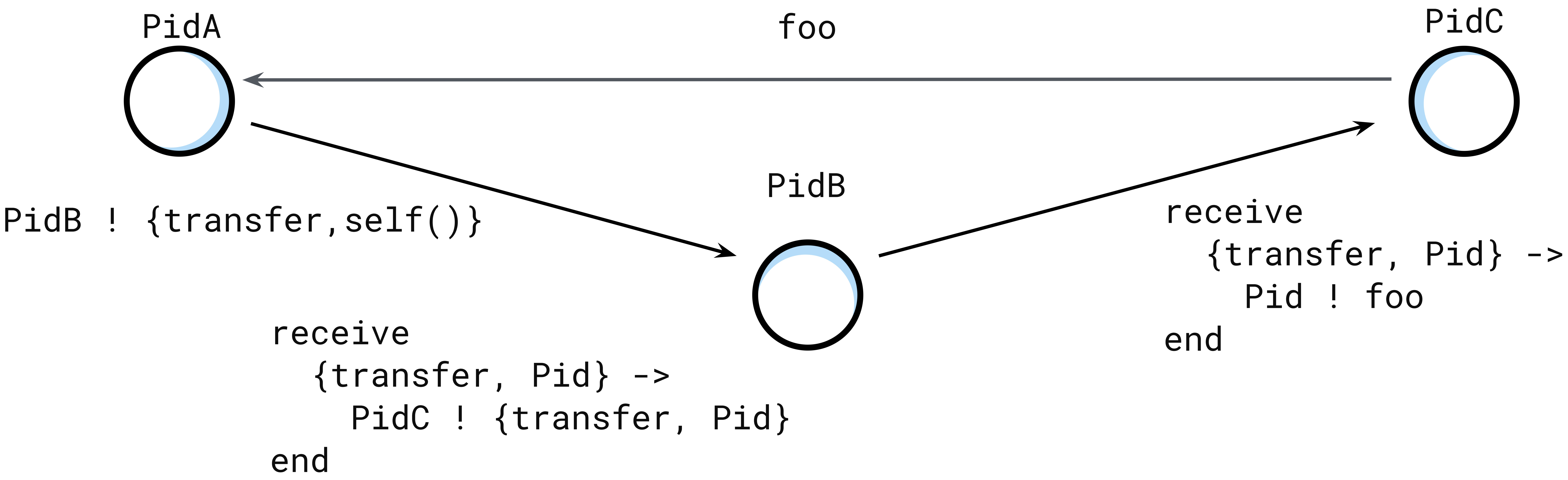
Pid2

Pid3 ! bar

```
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



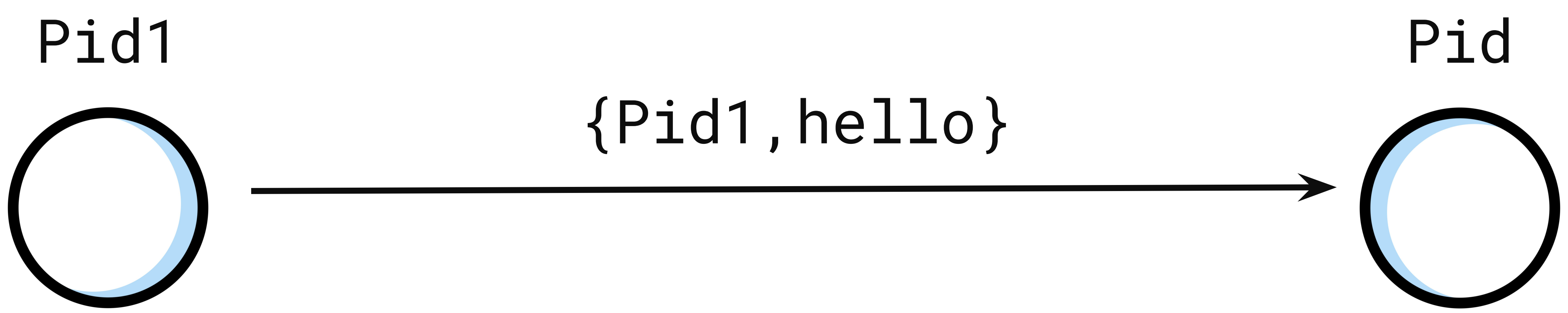
```
PidB ! {transfer,self()}
```

```
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 Pid and sends a message to PidC
- PidC receives the message and replies directly to PidA

Data in Messages: example



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

```
-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.
```

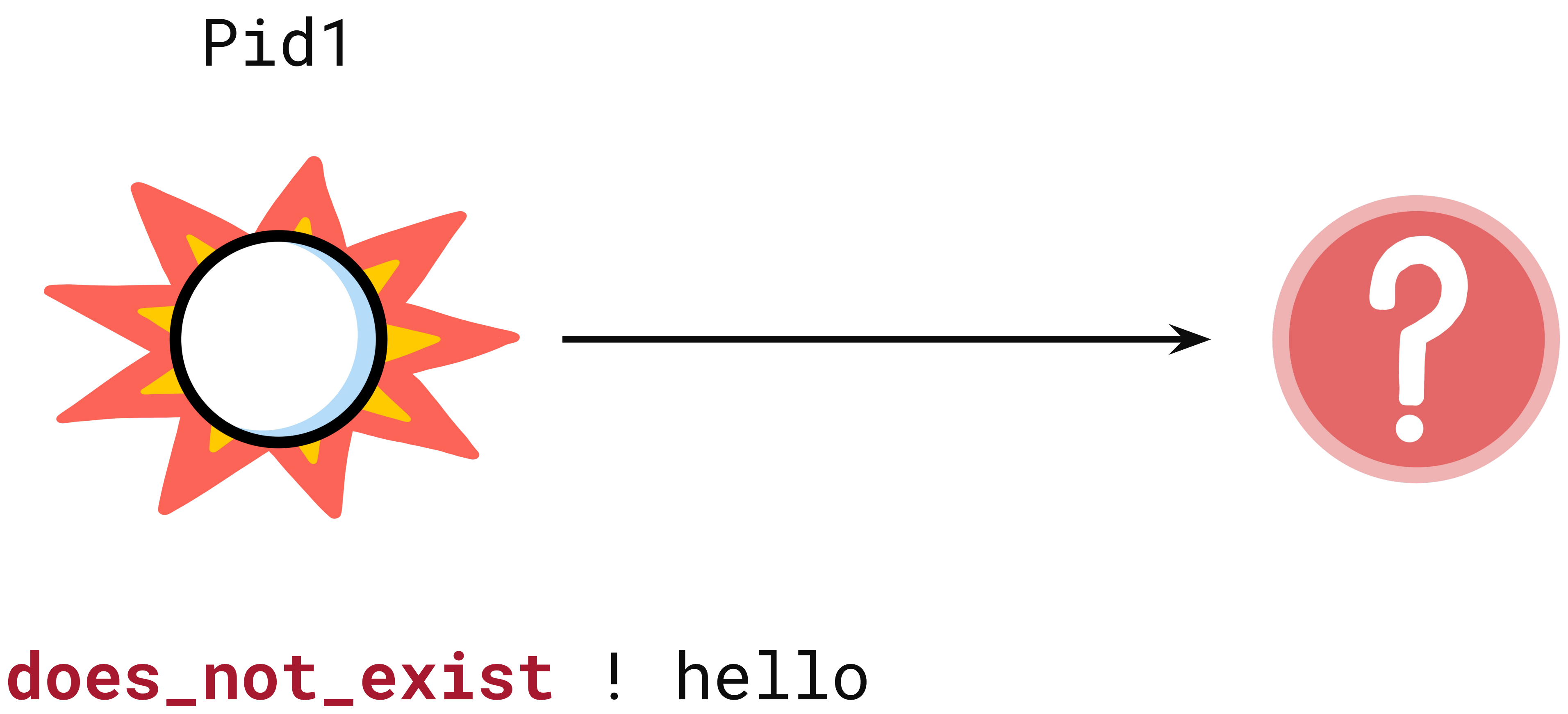
```
loop() ->
  receive
    {From, Msg} ->
      From ! {self(), Msg},
      loop();
    stop ->
      ok
  end.
```

Registered Processes

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

- Registers the process **Pid** with the name **Alias**, which has to be an atom
- 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**.

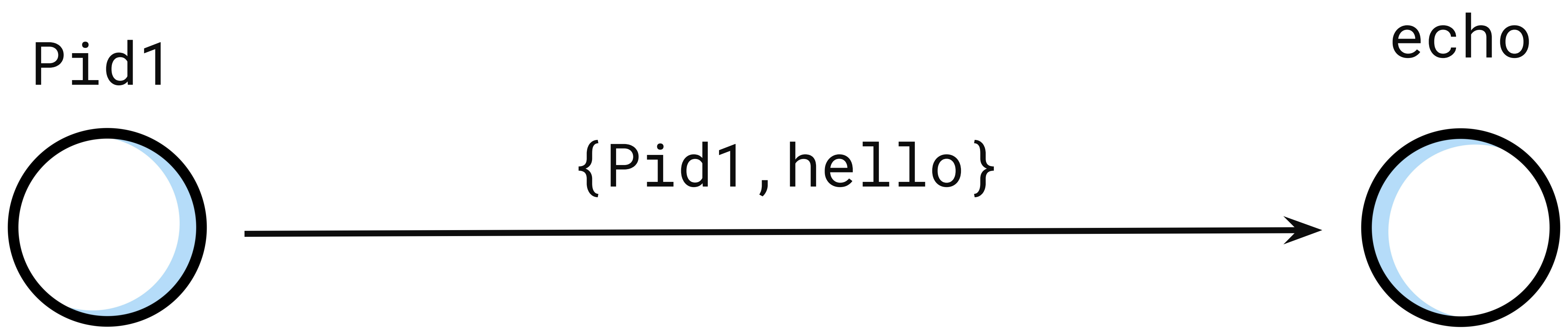
Message Passing



```
does_not_exist ! hello
```

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

Registered Processes



Pid1

{Pid1,hello}

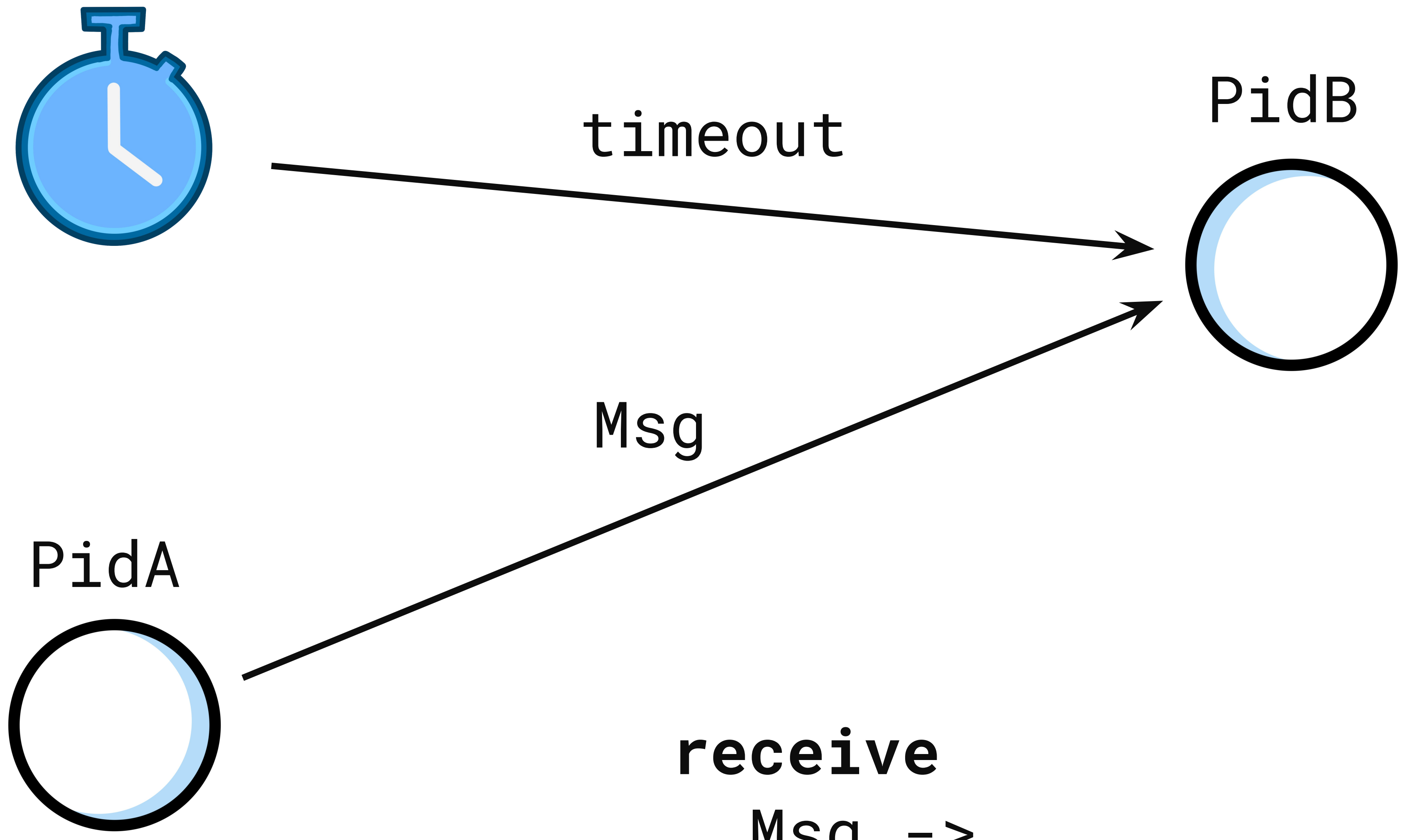
echo

echo ! {self(), hello}

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

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

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

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) ->
    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

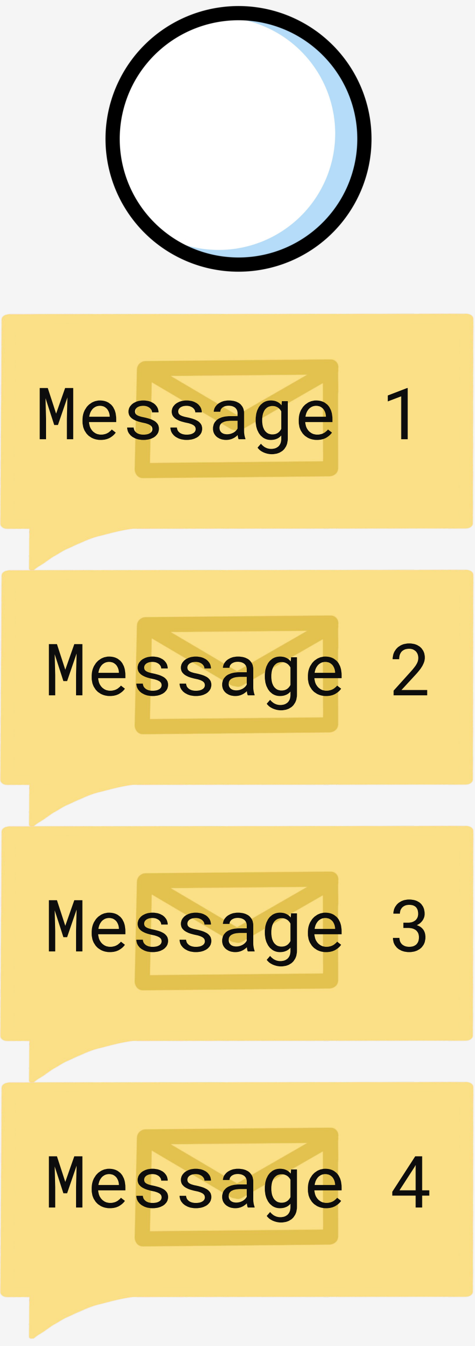
Timeouts

```
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**.

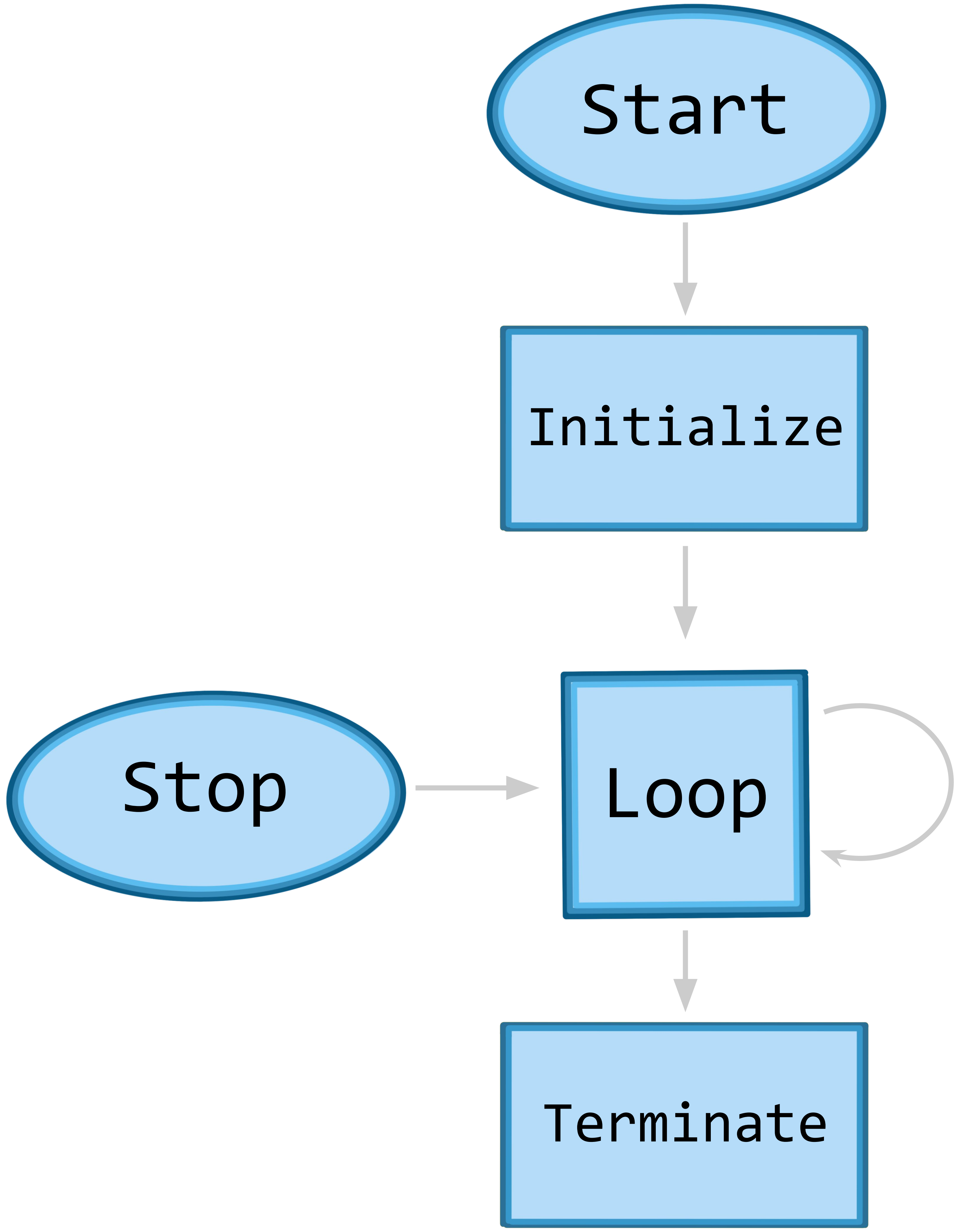
Termination

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).
```


Start

Initialize

Stop

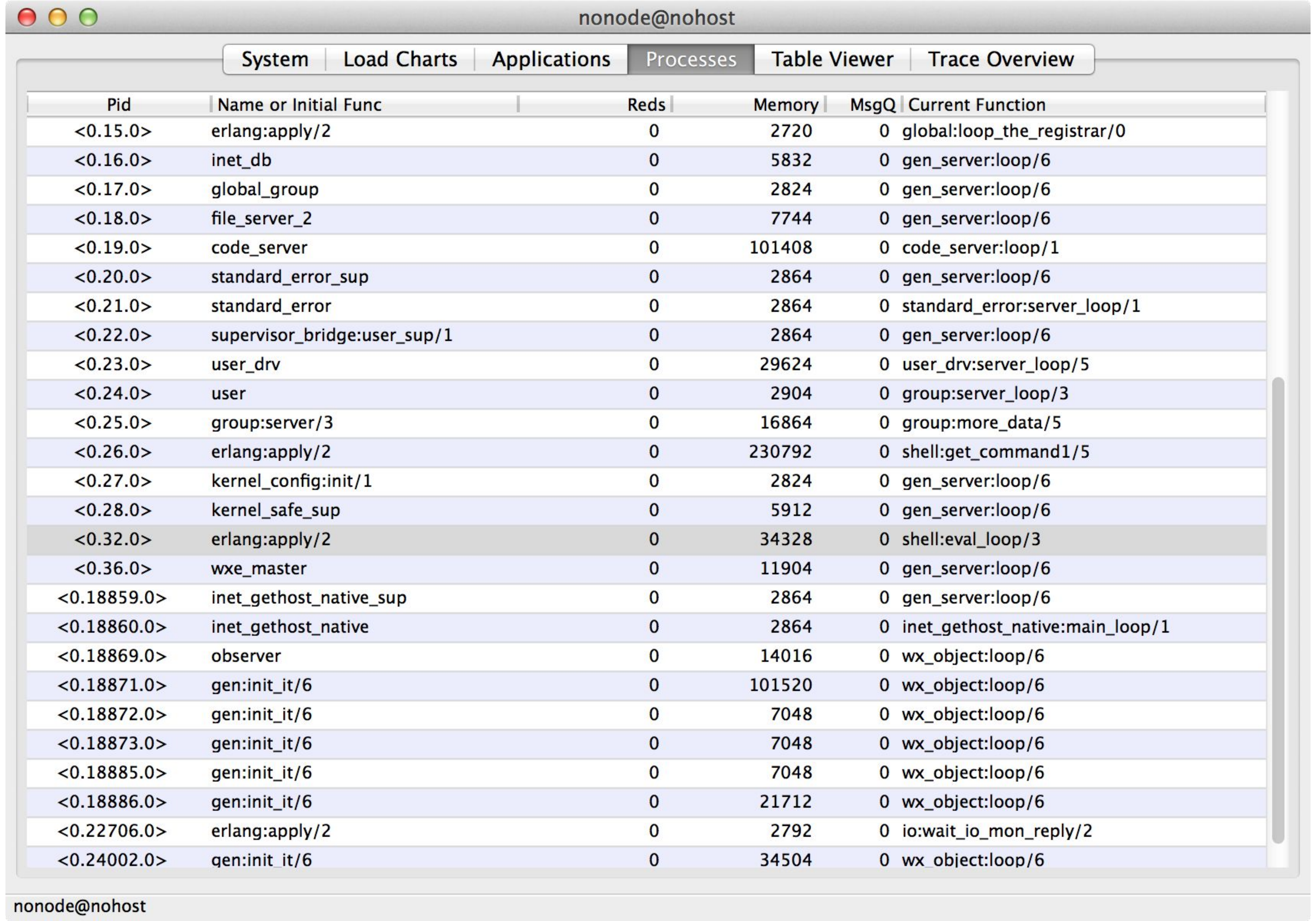
Loop

Terminate

Observer Processes

- A component of the observer graphical tool used to inspect the state of processes
    - Including processes in connected nodes
- Trace output for messages sent & received
- Trace output for process events such as spawn, exit and link
- Trace output for BIF and function calls
- Replaces the **pman** tool which was discontinued after R16

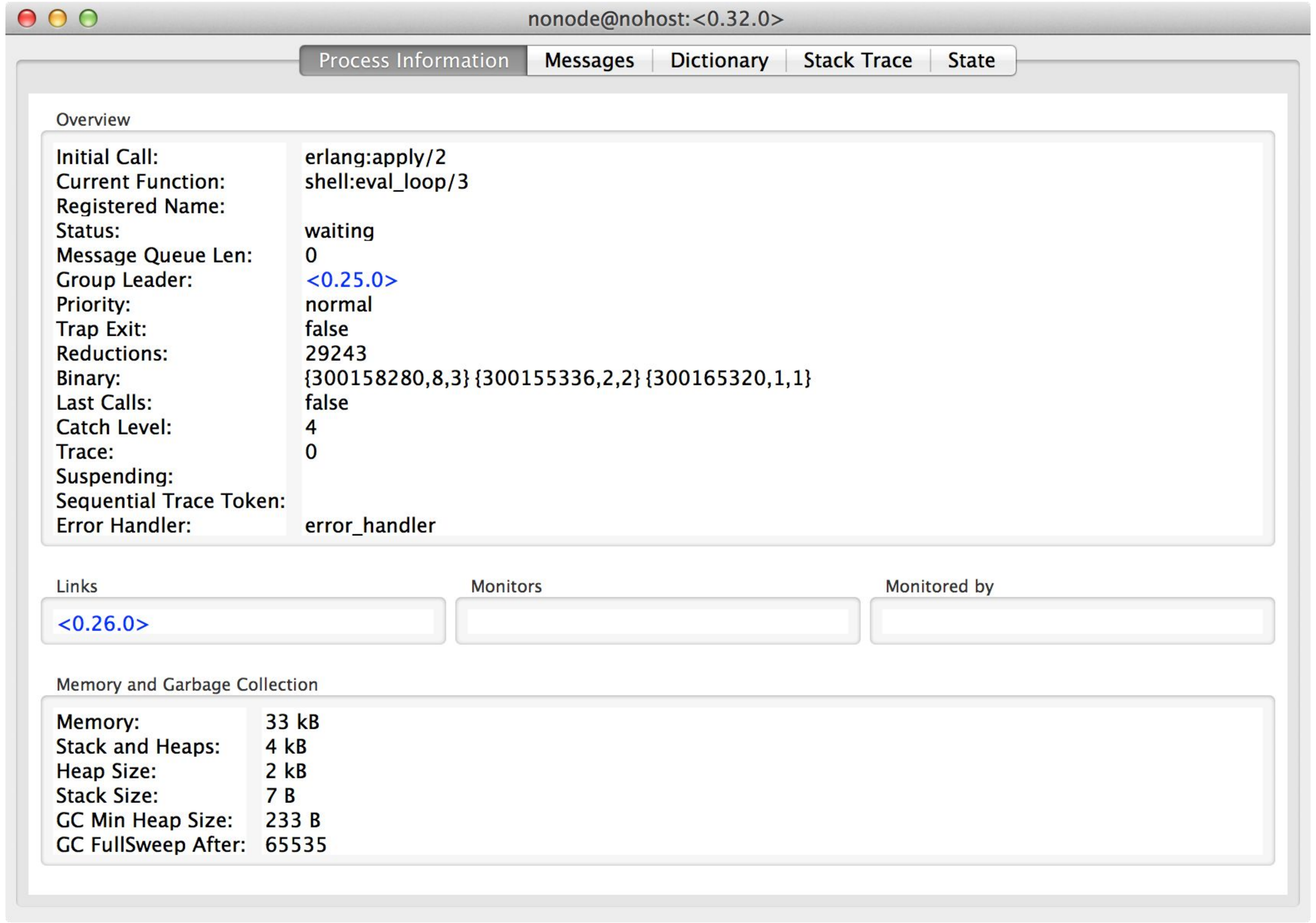
Observer Processes



Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.15.0>	erlang:apply/2	0	2720	0	global:loop_the_registrar/0
<0.16.0>	inet_db	0	5832	0	gen_server:loop/6
<0.17.0>	global_group	0	2824	0	gen_server:loop/6
<0.18.0>	file_server_2	0	7744	0	gen_server:loop/6
<0.19.0>	code_server	0	101408	0	code_server:loop/1
<0.20.0>	standard_error_sup	0	2864	0	gen_server:loop/6
<0.21.0>	standard_error	0	2864	0	standard_error:server_loop/1
<0.22.0>	supervisor_bridge:user_sup/1	0	2864	0	gen_server:loop/6
<0.23.0>	user_drv	0	29624	0	user_drv:server_loop/5
<0.24.0>	user	0	2904	0	group:server_loop/3
<0.25.0>	group:server/3	0	16864	0	group:more_data/5
<0.26.0>	erlang:apply/2	0	230792	0	shell:get_command1/5
<0.27.0>	kernel_config:init/1	0	2824	0	gen_server:loop/6
<0.28.0>	kernel_safe_sup	0	5912	0	gen_server:loop/6
<0.32.0>	erlang:apply/2	0	34328	0	shell:eval_loop/3
<0.36.0>	wxe_master	0	11904	0	gen_server:loop/6
<0.18859.0>	inet_gethost_native_sup	0	2864	0	gen_server:loop/6
<0.18860.0>	inet_gethost_native	0	2864	0	inet_gethost_native:main_loop/1
<0.18869.0>	observer	0	14016	0	wx_object:loop/6
<0.18871.0>	gen:init_it/6	0	101520	0	wx_object:loop/6
<0.18872.0>	gen:init_it/6	0	7048	0	wx_object:loop/6
<0.18873.0>	gen:init_it/6	0	7048	0	wx_object:loop/6
<0.18885.0>	gen:init_it/6	0	7048	0	wx_object:loop/6
<0.18886.0>	gen:init_it/6	0	21712	0	wx_object:loop/6
<0.22706.0>	erlang:apply/2	0	2792	0	io:wait_io_mon_reply/2
<0.24002.0>	gen:init_it/6	0	34504	0	wx_object:loop/6

- ➢ observer:start()

Observer Processes



Overview

Initial Call: erlang:apply/2
Current Function: shell:eval_loop/3
Registered Name:
Status: waiting
Message Queue Len: 0
Group Leader: <0.25.0>
Priority: normal
Trap Exit: false
Reductions: 29243
Binary: {300158280,8,3} {300155336,2,2} {300165320,1,1}
Last Calls: false
Catch Level: 4
Trace: 0
Suspending:
Sequential Trace Token:
Error Handler: error_handler

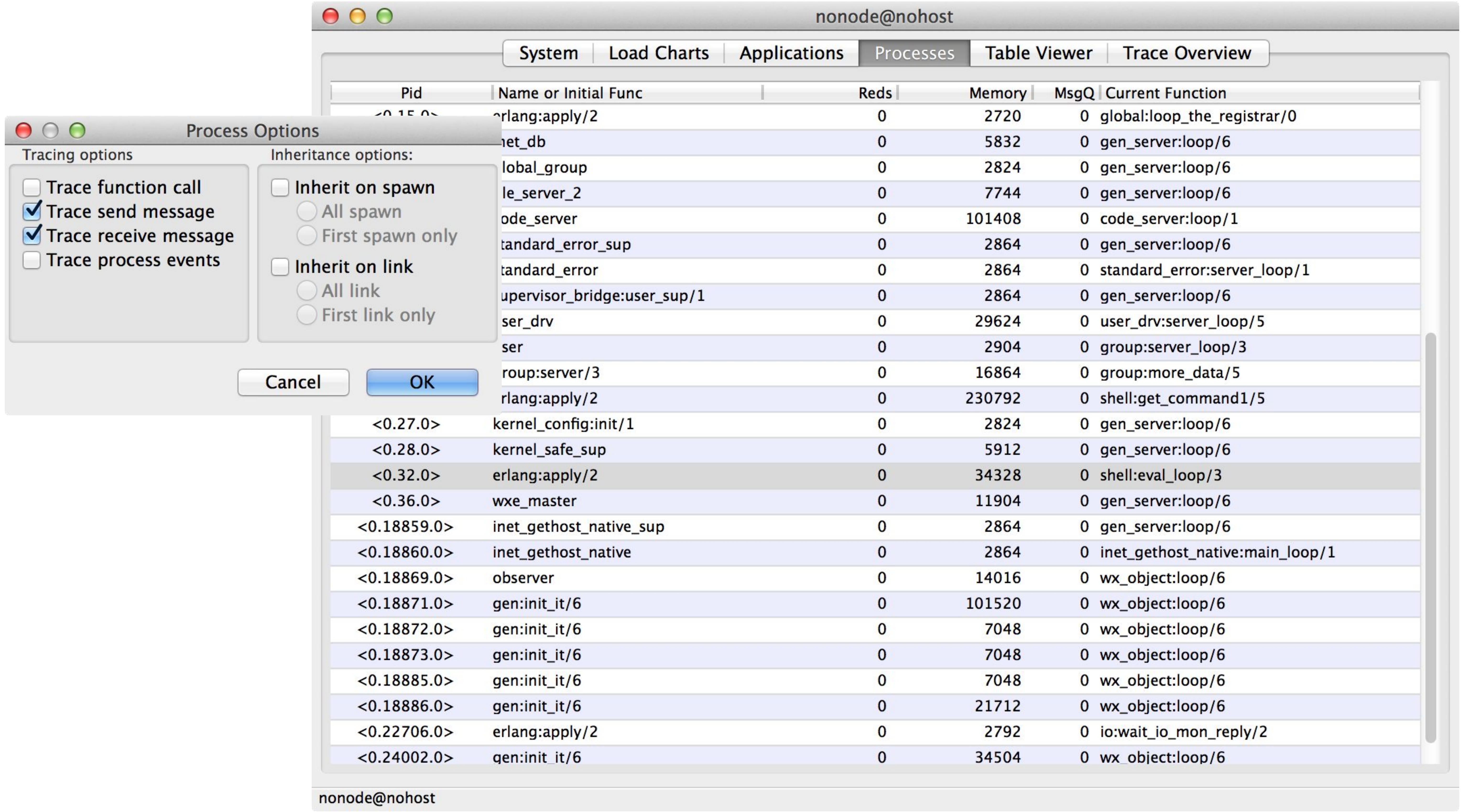
Links: <0.26.0>
Monitors:
Monitored by:

Memory and Garbage Collection

Memory: 33 kB
Stack and Heaps: 4 kB
Heap Size: 2 kB
Stack Size: 7 B
GC Min Heap Size: 233 B
GC FullSweep After: 65535

➢ Prints process information

Observer Processes



Process Options

Tracing options
- □ Trace function call
- ☑ Trace send message
- ☑ Trace receive message
- □ Trace process events

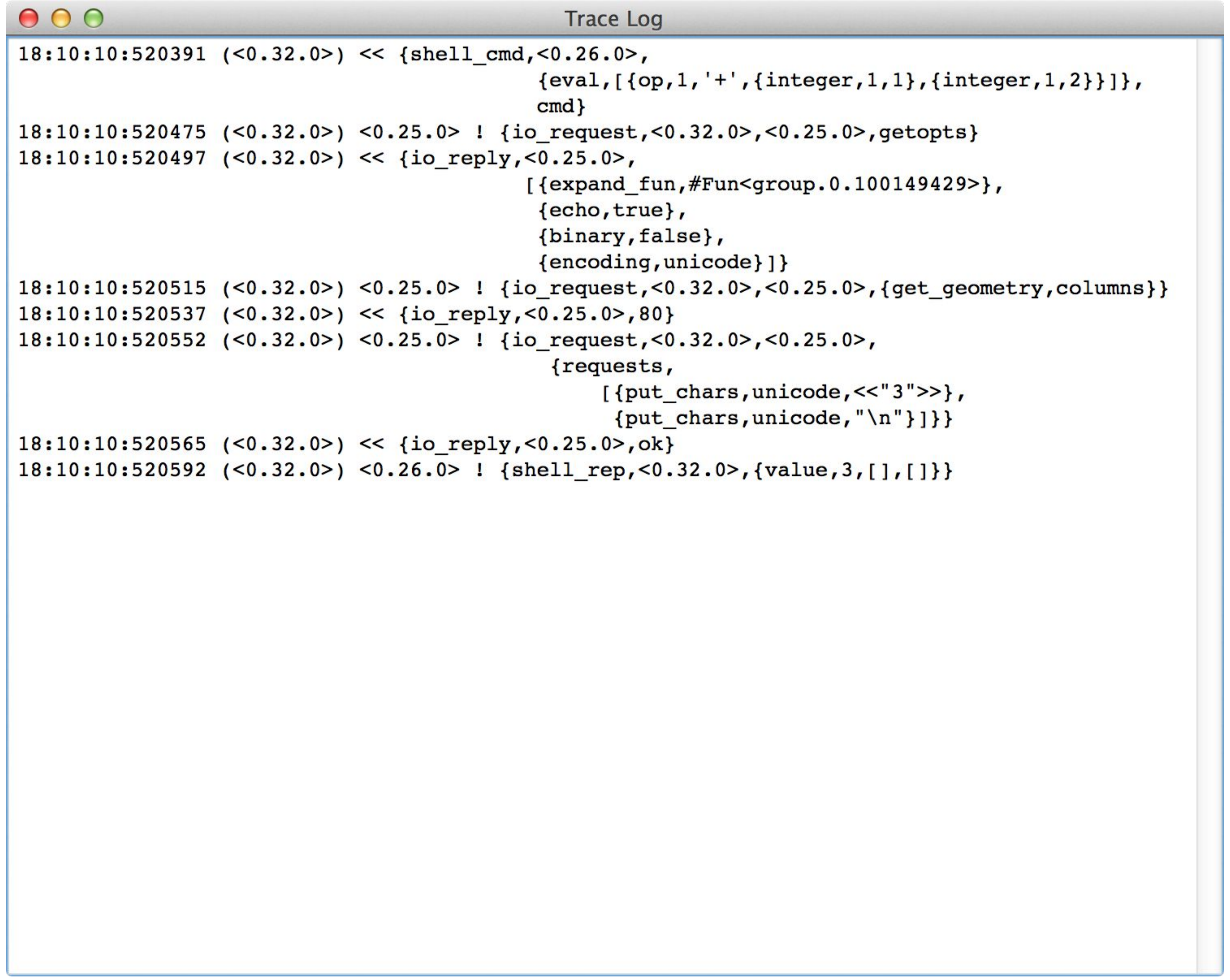
Inheritance options:
- □ Inherit on spawn
  - ○ All spawn
  - ○ First spawn only
- □ Inherit on link
  - ○ All link
  - ○ First link only

Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.15.0>	erlang:apply/2	0	2720	0	global:loop_the_registrar/0
	net_db	0	5832	0	gen_server:loop/6
	lobal_group	0	2824	0	gen_server:loop/6
	le_server_2	0	7744	0	gen_server:loop/6
	ode_server	0	101408	0	code_server:loop/1
	tandard_error_sup	0	2864	0	gen_server:loop/6
	tandard_error	0	2864	0	standard_error:server_loop/1
	upervisor_bridge:user_sup/1	0	2864	0	gen_server:loop/6
	ser_drv	0	29624	0	user_drv:server_loop/5
	ser	0	2904	0	group:server_loop/3
	roup:server/3	0	16864	0	group:more_data/5
	rlang:apply/2	0	230792	0	shell:get_command1/5
<0.27.0>	kernel_config:init/1	0	2824	0	gen_server:loop/6
<0.28.0>	kernel_safe_sup	0	5912	0	gen_server:loop/6
<0.32.0>	erlang:apply/2	0	34328	0	shell:eval_loop/3
<0.36.0>	wxe_master	0	11904	0	gen_server:loop/6
<0.18859.0>	inet_gethost_native_sup	0	2864	0	gen_server:loop/6
<0.18860.0>	inet_gethost_native	0	2864	0	inet_gethost_native:main_loop/1
<0.18869.0>	observer	0	14016	0	wx_object:loop/6
<0.18871.0>	gen:init_it/6	0	101520	0	wx_object:loop/6
<0.18872.0>	gen:init_it/6	0	7048	0	wx_object:loop/6
<0.18873.0>	gen:init_it/6	0	7048	0	wx_object:loop/6
<0.18885.0>	gen:init_it/6	0	7048	0	wx_object:loop/6
<0.18886.0>	gen:init_it/6	0	21712	0	wx_object:loop/6
<0.22706.0>	erlang:apply/2	0	2792	0	io:wait_io_mon_reply/2
<0.24002.0>	gen:init_it/6	0	34504	0	wx_object:loop/6

nonode@nohost

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

Observer Processes



```
18:10:10:520391 (<0.32.0>) << {shell_cmd,<0.26.0>,
                                         {eval,[{op,1,'+',{integer,1,1},{integer,1,2}}]},
                                         cmd}
18:10:10:520475 (<0.32.0>) <0.25.0> ! {io_request,<0.32.0>,<0.25.0>,getopts}
18:10:10:520497 (<0.32.0>) << {io_reply,<0.25.0>,
                                        [{expand_fun,#Fun<group.0.100149429>},
                                         {echo,true},
                                         {binary,false},
                                         {encoding,unicode}]}
18:10:10:520515 (<0.32.0>) <0.25.0> ! {io_request,<0.32.0>,<0.25.0>,{get_geometry,columns}}
18:10:10:520537 (<0.32.0>) << {io_reply,<0.25.0>,80}
18:10:10:520552 (<0.32.0>) <0.25.0> ! {io_request,<0.32.0>,<0.25.0>,
                                          {requests,
                                              [{put_chars,unicode,<<"3">>},
                                               {put_chars,unicode,"\n"}]}}
18:10:10:520565 (<0.32.0>) << {io_reply,<0.25.0>,ok}
18:10:10:520592 (<0.32.0>) <0.26.0> ! {shell_rep,<0.32.0>,{value,3,[],[]}}
```

- ➢ Start tracing in the Trace Overview
- ➢ Prints trace information

Summary: concurrent Erlang

- ▶ Creating Processes
- ▶ Message Passing
- ▶ Receiving Messages
- ▶ Data in Messages
- ▶ Registered Processes
- ▶ Timeouts
- ▶ More on Processes
- ▶ Observer Processes