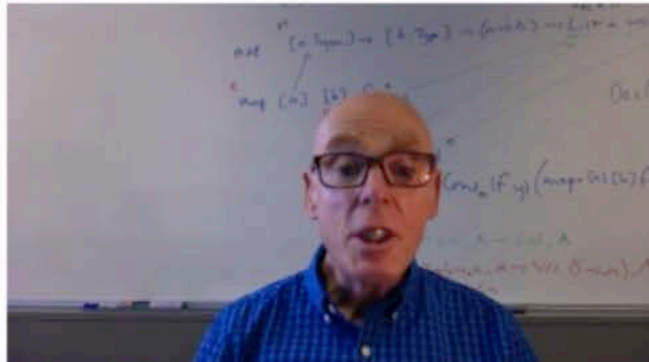
University of Kent

Message-passing concurrency



University of Kent

Erlang rationale?

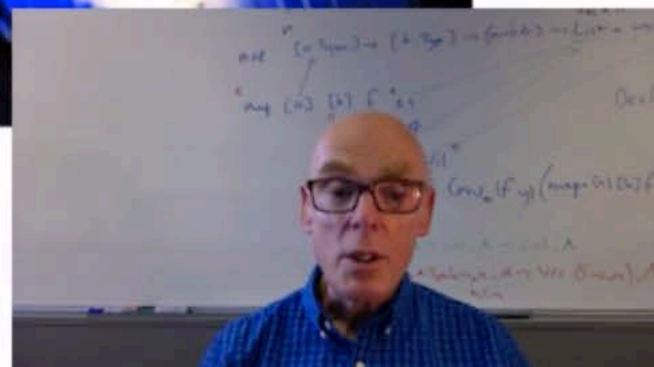
Designed by Ericsson for telecoms systems.

These must be …

… concurrent,

… high-availability, and

… fault-tolerant.



University of Kent

Erlang rationale?

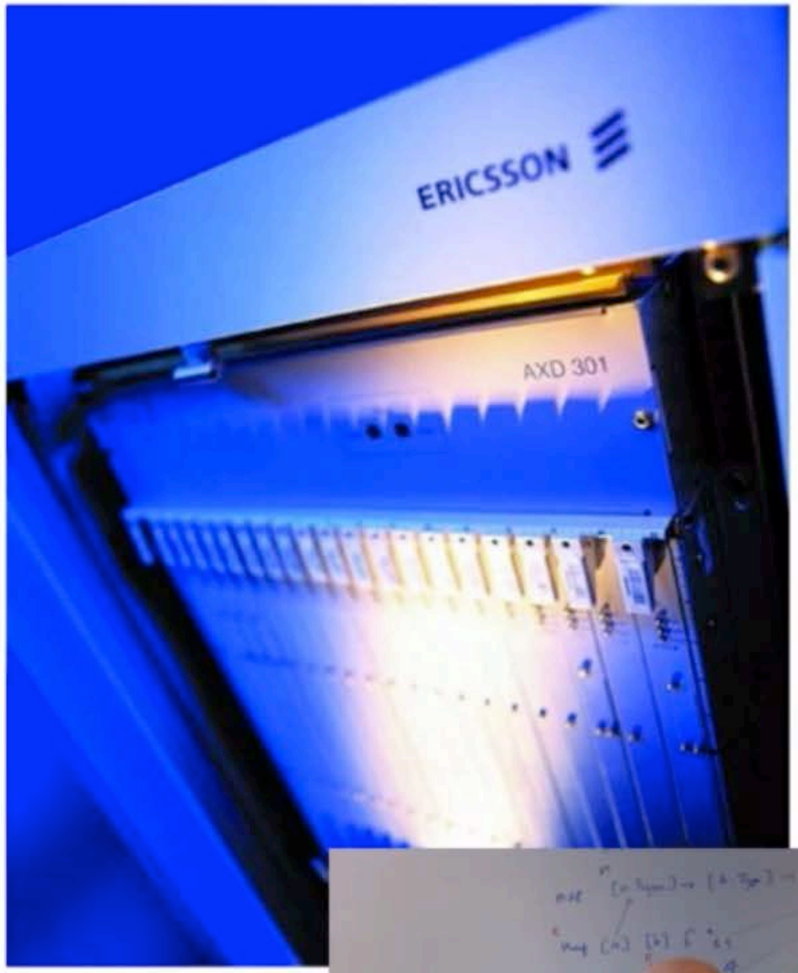
Used today by WhatsApp!, Cisco, Bet365, …

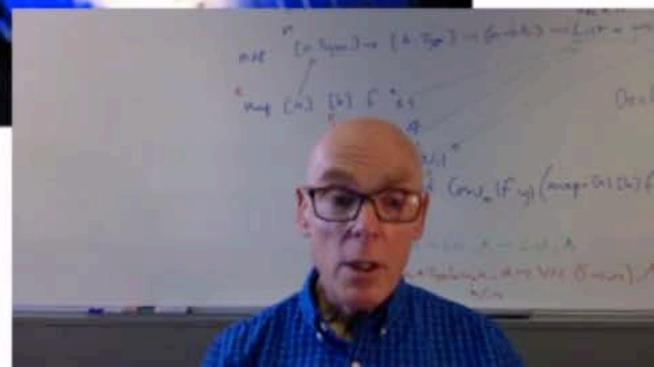
Because it is …

… concurrent,

… high-availability, and

… fault-tolerant.





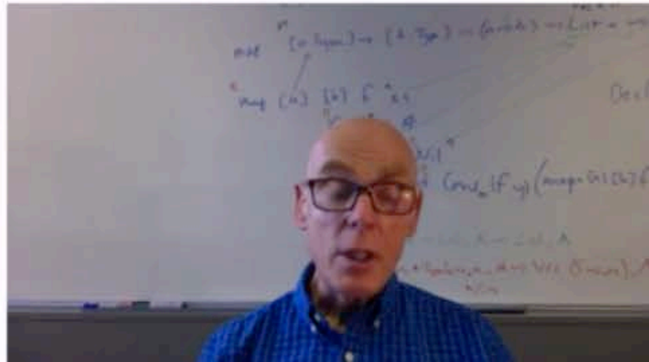
University of Kent

Based on functional programming

Functional is simpler: values and not references, pointers etc.

Functional supports high-level patterns e.g. map/reduce.

With no side-effects ("immutable data") algorithms clearer.



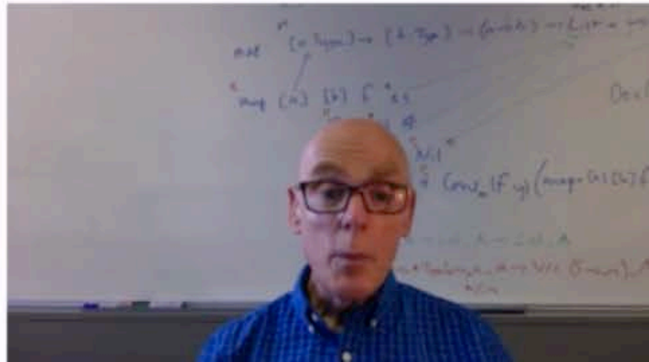
University of Kent

Immutability as a design pattern

Structures whose state doesn't change …

… if you want a different structure, create a new one.

Thread-safe programming … safe caching and sharing … consistency



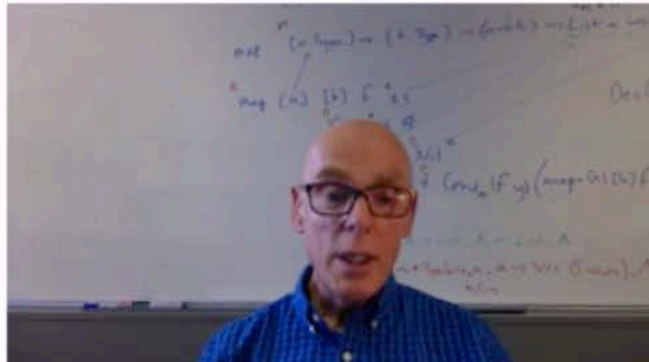
University of Kent

A pragmatic approach to side-effects

At its core, Erlang is functional …

…it does allow some side-effects (e.g. communication) …

… but not others (no updatable Java-style variables).



University of Kent

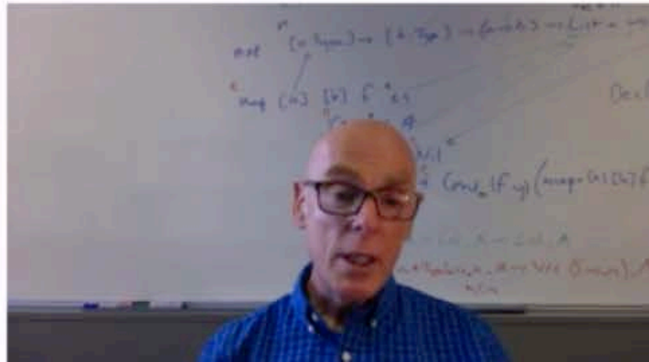
Concurrency

Modularity: different "processes" in different processes.

Robustness: "let it fail" and others processes sort it out.

Concurrency for design (independence) …

… versus parallelism for efficiency (happen at same time).



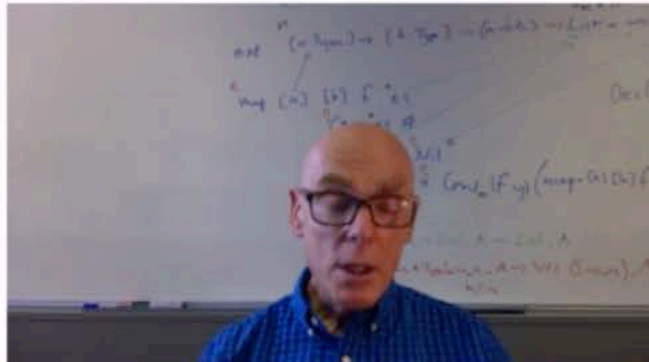
University of Kent

Message-passing concurrency

Different concurrent processes "share nothing" …

…and only communicate by passing messages.

No worries about shared state, thread safety, …



University of Kent

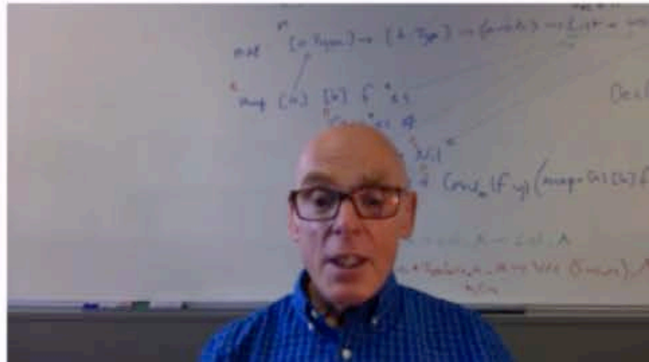
A beautiful design

Mailboxes and message handling

Process errors and trapping exits

OTP generics

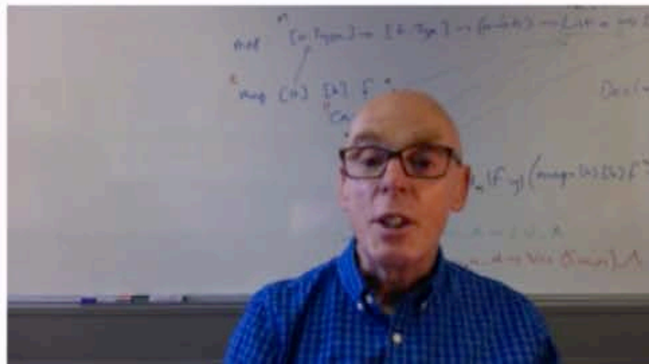
Concurrency and distribution




University of
Kent

University of Kent

Processes and messages



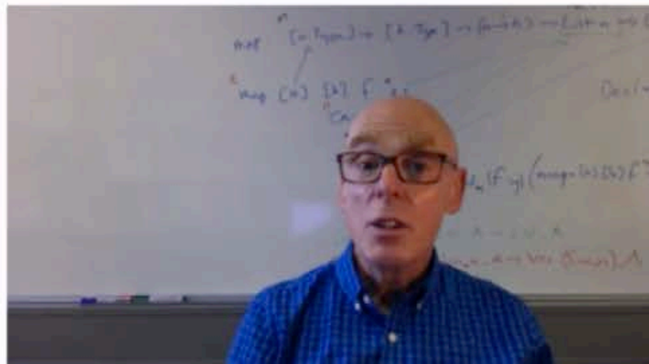
University of Kent

Message-passing concurrency

Different concurrent processes "share nothing" …

…and only communicate by passing messages.

No worries about shared state, thread safety, …



University of Kent

The "actor model"

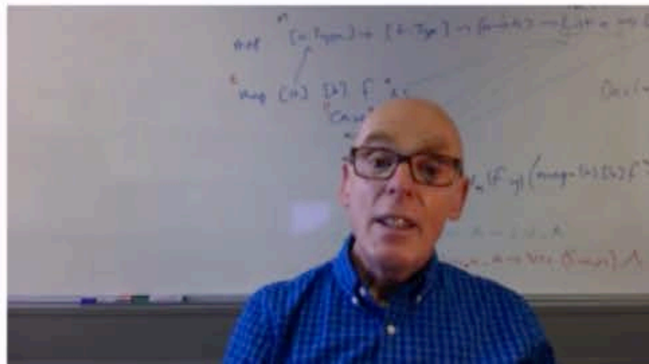
Carl Hewitt's model for the foundations of computation …

… everything is an actor

… actors communicate by messages

… actors can create other actors

… and change behaviour according to messages.

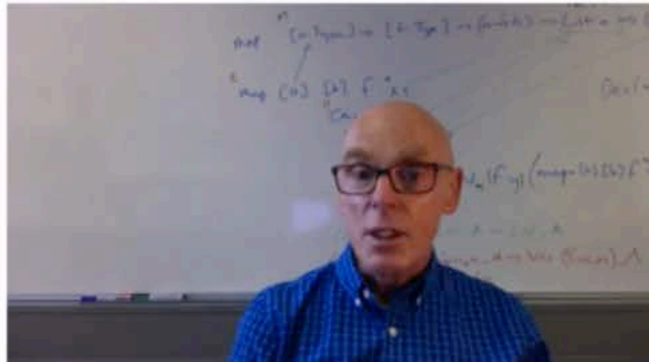


University of Kent

The design of Erlang's concurrency

Erlang designed to build solutions in a particular problem space.

Choices to support that … the roads not chosen.



University of Kent

Processes *versus* threads

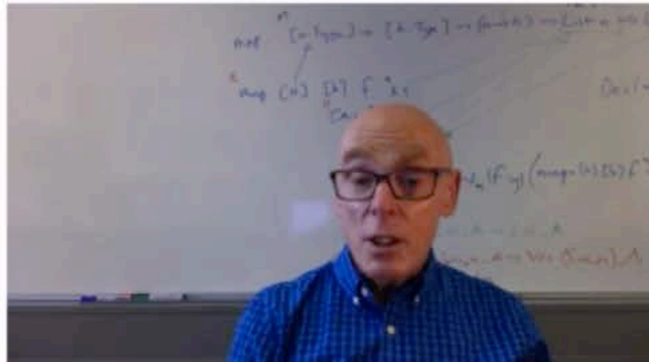
Processes *share nothing* with each other.

In the virtual machine, not the operating system … like "green threads" but …

…more lightweight than threads

…more numerous

… more control: stack, heap and garbage collection per process.



University of Kent

Synchronous *versus* asynchronous

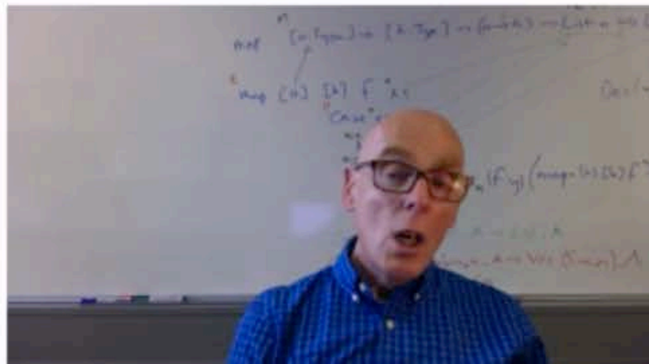
Message send and message receive are decoupled.

… avoids a common source of deadlock

…directly supports processes distributed across a network

…supports more complex message handling

…can simulate synchrony through send/receive protocol



University of Kent

Identifiers *versus* channels

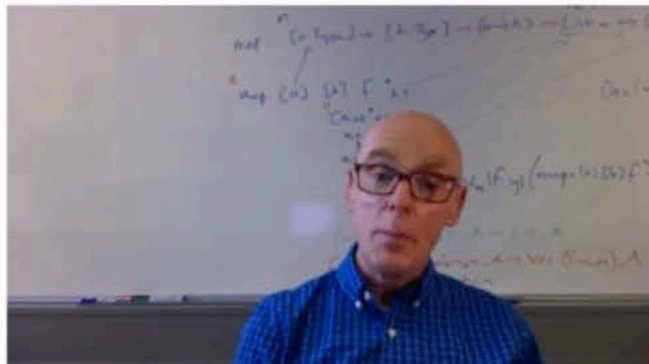
Messages are sent to a particular process id (Pid) …

… rather than along a channel between processes.

The communication structure is less evident in the program …

… but supports dynamic systems more readily.

Named processes give more visible long-lived structures.



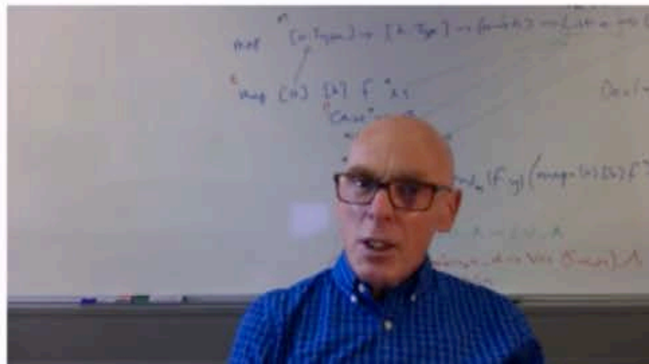
University of Kent

Concurrency *versus* parallelism

Concurrency means the possibility of running independently, perhaps at the same time.

On a single processing element, concurrent processes timeshare, mediated by a scheduler.

On a multicore processor, there is – potentially at least – concurrent activity on each core, with each scheduled separately.



University of Kent

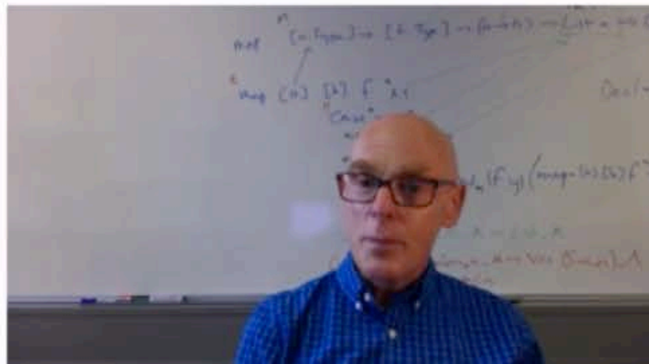
A beautiful design

Mailboxes and message handling

Process errors and trapping exits

OTP generics

Concurrency and distribution




University of
Kent

University of Kent

Processes and messages in practice

University of Kent

Erlang concurrency in a nutshell

`spawn` – create a process

`!` – send a message

`self()` – give the Process Identifier (Pid) of a process

`receive` – handle a message

University of Kent

What is a process?

A process is a separate computation

… running in its own space

… time-sharing with other processes

A process runs an Erlang function

… and terminates when / if that function terminates

University of Kent

`spawn` – create a process

```
spawn(Mod,Fun,Args)
```

`Mod` is a module,

`Fun` is a function in `Mod`,

`Args` is a list of arguments to pass to `Fun` at startup.

For example:

```
spawn(foo,bar,[])
```

University of Kent

`spawn` returns a process id

A call like

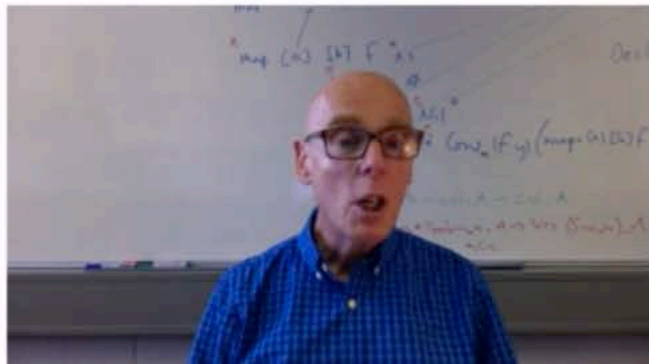
```
spawn(Mod,Fun,Args)
```

returns the process id (Pid)
of the process created.

It is typically captured like this:

```
Proc = spawn(foo,bar,[])
```

```
-module(foo).
-export([bar/0]).

bar() ->
  timer:wait(500),
  io:format("bar started~n"),
  io:format("bar working~n"),
  io:format("bar finished~n").
```



Terminal Shell Edit View Window Help

70% 13:12

foo.erl

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

```
-module(foo).
-export([bar/0,bar/1,baz/0,bazz/0]).

bar() ->
  timer:sleep(500),
  io:format("bar started~n"),
  io:format("bar working~n"),
  io:format("bar finished~n").

bar(Pid) ->
  Pid ! "bar started~n",
  Pid ! "bar working~n",
  Pid ! "bar finished~n".

baz() ->
  receive
    Msg ->
      io:format("got: ~s~n", [Msg])
  end,
  baz().

bazz() ->
  receive
    stop ->
      io:format("stopped~n");
    Msg ->
      io:format("got: ~s~n", [Msg]),
      bazz()
  end.
```

-:--- foo.erl All (29,0) (Erlang EXT Flymake)

No further undo information

simonthompson — xterm — bash

%

University of Kent

Terminal Shell Edit View Window Help

foo.erl

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

```
-module(foo).
-export([bar/0,bar/1,baz/0,bazz/0]).

bar() ->
  timer:sleep(500),
  io:format("bar started~n"),
  io:format("bar working~n"),
  io:format("bar finished~n").

bar(Pid) ->
  Pid ! "bar started~n",
  Pid ! "bar working~n",
  Pid ! "bar finished~n".

baz() ->
  receive
    Msg ->
      io:format("got: ~s~n", [Msg])
  end,
  baz().

bazz() ->
  receive
    stop ->
      io:format("stopped~n");
    Msg ->
      io:format("got: ~s~n", [Msg]),
      bazz()
  end.
```

-:--- foo.erl All (29,0) (Erlang EXT Flymake)

No further undo information

simonthompson — xterm — beam.smp -- -root /usr/local/lib/erlang -progname erl -- -home ~ -- ▸ erl_child_setup

```
% erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:8:8] [async-threads:10]
[hipe] [kernel-poll:false]

Eshell V8.0  (abort with ^G)
1> c(foo).
{ok,foo}
2> spawn(foo,bar,[]).
<0.64.0>
bar started
bar working
bar finished
3> spawn(foo,bar,[]).
<0.66.0>
bar started
bar working
bar finished
4>
```

University of Kent

`!` – send a message

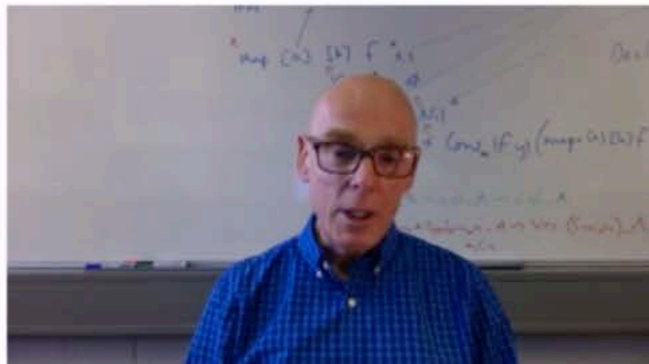
`Pid ! Msg`

sends the message `Msg` to the process with process id `Pid` ...

We say "sends `Msg` to `Pid`".

The effect is to put `Msg` into the mailbox of process `Pid`.

```
-module(foo).
-export([bar/1]).

bar(Pid) ->
  Pid ! "bar started~n",
  Pid ! "bar working~n",
  Pid ! "bar finished~n".
```

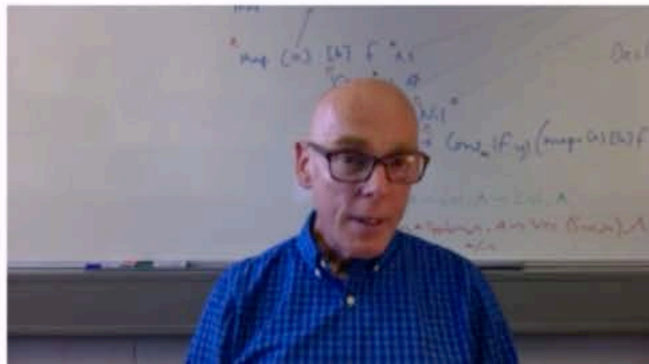


University of Kent

`self()` – what's my Pid?

`self()` gives the `Pid` of the process in which it's called.

In particular, we can get the Pid of the Erlang shell process by calling `self()` there.



Terminal Shell Edit View Window Help

foo.erl

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

```
-module(foo).
-export([bar/0,bar/1,baz/0,bazz/0]).

bar() ->
  timer:sleep(500),
  io:format("bar started~n"),
  io:format("bar working~n"),
  io:format("bar finished~n").

bar(Pid) ->
  Pid ! "bar started~n",
  Pid ! "bar working~n",
  Pid ! "bar finished~n".

baz() ->
  receive
    Msg ->
      io:format("got: ~s~n", [Msg])
  end,
  baz().

bazz() ->
  receive
    stop ->
      io:format("stopped~n");
    Msg ->
      io:format("got: ~s~n", [Msg]),
      bazz()
  end.
```

-:--- foo.erl All (29,0) (Erlang EXT Flymake)

No further undo information

simonthompson — xterm — beam.smp -- -root /usr/local/lib/erlang -progname erl -- -home ~ -- ▸ erl_child_setup

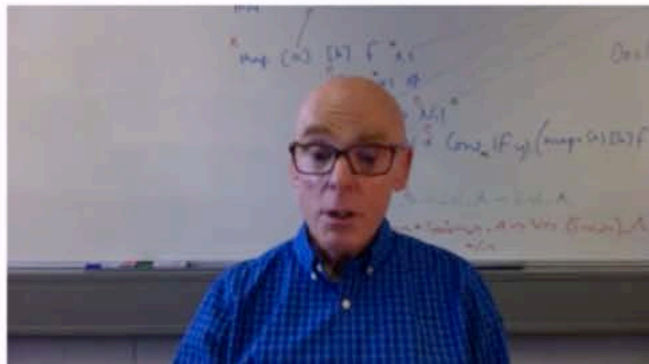
```
% erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:8:8] [async-threads:10]
[hipe] [kernel-poll:false]

Eshell V8.0  (abort with ^G)
1> c(foo).
{ok,foo}
2> spawn(foo,bar,[]).
<0.64.0>
bar started
bar working
bar finished
3> spawn(foo,bar,[]).
<0.66.0>
bar started
bar working
bar finished
4> spawn(foo,bar,[self()]).
<0.68.0>
5> flush().
Shell got "bar started~n"
Shell got "bar working~n"
Shell got "bar finished~n"
ok
6>
```

University of Kent

`receive` – receive a message

```
receive
  Msg -> … handle Msg …
end
```

takes the message `Msg` out of the mailbox and handles it.



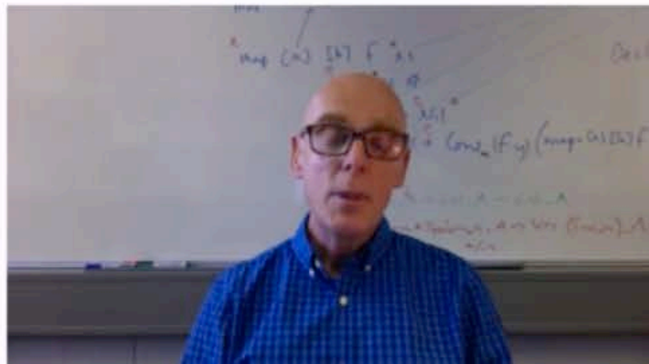
University of Kent

`receive` – receive a message

```
receive
  Msg -> … handle Msg …
end
```

takes the message `Msg` out of the mailbox and handles it.

```
-module(foo).
-export([baz/0]).

baz() ->
  receive
    Msg ->
      io:format("got: ~s~n",[Msg])
  end,
  baz().
```

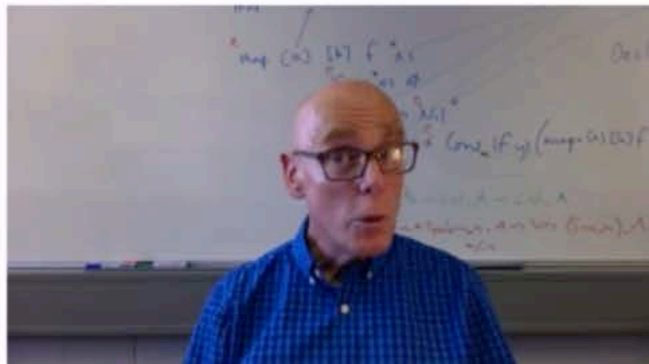


University of Kent

`receive` – receive a message

The general `receive` construct lets us pattern match on messages and choose between alternatives.

Just like a `case` statement.



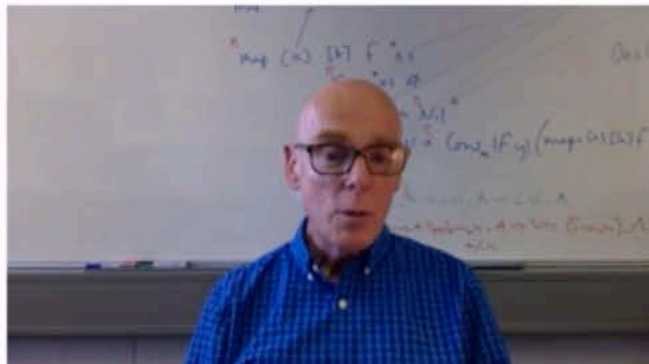
University of Kent

`receive` – receive a message

The general `receive` construct lets us pattern match on messages and choose between alternatives.

Just like a `case` statement.

```
-module(foo).
-export([bazz/0]).

bazz() ->
  receive
    stop ->
      io:format("stopped~n");
    Msg ->
      io:format("got: ~s~n",[Msg]),
      bazz()
  end.
```



Terminal Shell Edit View Window Help

foo.erl

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

```
-module(foo).
-export([bar/0,bar/1,baz/0,bazz/0]).

bar() ->
  timer:sleep(500),
  io:format("bar started~n"),
  io:format("bar working~n"),
  io:format("bar finished~n").

bar(Pid) ->
  Pid ! "bar started~n",
  Pid ! "bar working~n",
  Pid ! "bar finished~n".

baz() ->
  receive
    Msg ->
      io:format("got: ~s~n", [Msg])
  end,
  baz().

bazz() ->
  receive
    stop ->
      io:format("stopped~n");
    Msg ->
      io:format("got: ~s~n", [Msg]),
      bazz()
  end.
```

-:--- foo.erl All (29,0) (Erlang EXT Flymake)

No further undo information

simonthompson — xterm — beam.smp -- -root /usr/local/lib/erlang -progname erl -- -home ~ -- ▸ erl_child_setup

```
% erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:8:8] [async-threads:10]
[hipe] [kernel-poll:false]

Eshell V8.0  (abort with ^G)
1> c(foo).
{ok,foo}
2> spawn(foo,bar,[]).
<0.64.0>
bar started
bar working
bar finished
3> spawn(foo,bar,[]).
<0.66.0>
bar started
bar working
bar finished
4> spawn(foo,bar,[self()]).
<0.68.0>
5> flush().
Shell got "bar started~n"
Shell got "bar working~n"
Shell got "bar finished~n"
ok
6> Bazz
```

Terminal Shell Edit View Window Help

foo.erl

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

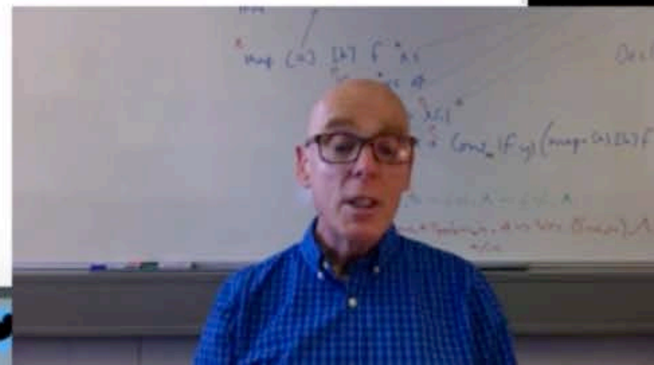
```
-module(foo).
-export([bar/0,bar/1,baz/0,bazz/0]).

bar() ->
  timer:sleep(500),
  io:format("bar started~n"),
  io:format("bar working~n"),
  io:format("bar finished~n").

bar(Pid) ->
  Pid ! "bar started~n",
  Pid ! "bar working~n",
  Pid ! "bar finished~n".

baz() ->
  receive
    Msg ->
      io:format("got: ~s~n", [Msg])
  end,
  baz().

bazz() ->
  receive
    stop ->
      io:format("stopped~n");
    Msg ->
      io:format("got: ~s~n", [Msg]),
      bazz()
  end.
```

-:--- foo.erl All (29,0) (Erlang EXT Flymake)

No further undo information

simonthompson — xterm — beam.smp -- -root /usr/local/lib/erlang -progname erl -- -home ~ -- ▸ erl_child_setup

```
% erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:8:8] [async-threads:10]
[hipe] [kernel-poll:false]

Eshell V8.0  (abort with ^G)
1> c(foo).
{ok,foo}
2> spawn(foo,bar,[]).
<0.64.0>
bar started
bar working
bar finished
3> spawn(foo,bar,[]).
<0.66.0>
bar started
bar working
bar finished
4> spawn(foo,bar,[self()]).
<0.68.0>
5> flush().
Shell got "bar started~n"
Shell got "bar working~n"
Shell got "bar finished~n"
ok
6> Bazz = spawn(foo,bazz,[]).
<0.71.0>
7> Bazz ! hello.
got: hello
hello
8> Bazz ! ola .
got: ola
ola
9> Bazz ! stop.
stopped
stop
10>
```



Terminal Shell Edit View Window Help

foo.erl

New Open Recent Revert Save Print Undo Redo Cut Copy Paste Search Preferences

```
-module(foo).
-export([bar/0,bar/1,baz/0,bazz/0]).

bar() ->
  timer:sleep(500),
  io:format("bar started~n"),
  io:format("bar working~n"),
  io:format("bar finished~n").

bar(Pid) ->
  Pid ! "bar started~n",
  Pid ! "bar working~n",
  Pid ! "bar finished~n".

baz() ->
  receive
    Msg ->
      io:format("got: ~s~n", [Msg])
  end,
  baz().

bazz() ->
  receive
    stop ->
      io:format("stopped~n");
    Msg ->
      io:format("got: ~s~n", [Msg]),
      bazz()
  end.
```

-:--- foo.erl All (29,0) (Erlang EXT Flymake)

No further undo information

simonthompson — xterm — beam.smp -- -root /usr/local/lib/erlang -progname erl -- -home ~ -- ▸ erl_child_setup

```
% erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:8:8] [async-threads:10]
[hipe] [kernel-poll:false]

Eshell V8.0  (abort with ^G)
1> c(foo).
{ok,foo}
2> spawn(foo,bar,[]).
<0.64.0>
bar started
bar working
bar finished
3> spawn(foo,bar,[]).
<0.66.0>
bar started
bar working
bar finished
4> spawn(foo,bar,[self()]).
<0.68.0>
5> flush().
Shell got "bar started~n"
Shell got "bar working~n"
Shell got "bar finished~n"
ok
6> Bazz = spawn(foo,bazz,[]).
<0.71.0>
7> Bazz ! hello.
got: hello
hello
8> Bazz ! ola .
got: ola
ola
9> Bazz ! stop.
stopped
stop
10> Bazz ! ola.
ola
11>
```

University of Kent

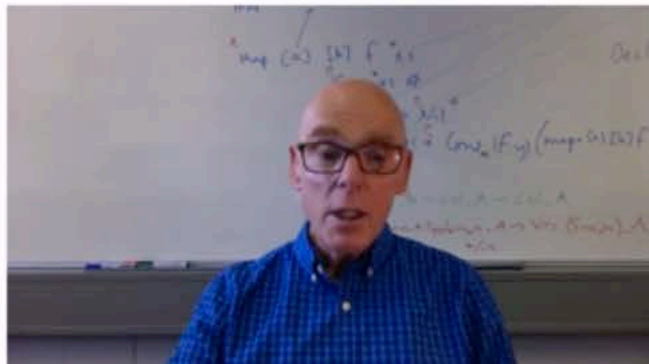
Erlang concurrency in a nutshell

`spawn` – create a process

`!` – send a message

`self()` – give the Process Identifier (Pid) of a process

`receive` – handle a message




University of
Kent

Joe Armstrong

EXPERT SYSTEM DEVELOPER, AND ONE OF THE
CREATORS OF ERLANG AT ERICSSON

University of
Kent

University of Kent

```
area({square,X}) ->
    X*X;
area({rectangle,X,Y}) ->
    X*Y.
```

University of Kent

```
{square,X} ->
    X*X;
{rectangle,X,Y} ->
    X*Y.
```

```
area({square,X}) ->
    X*X;
area({rectangle,X,Y}) ->
    X*Y.
```

```
area() ->
    receive
      {square,X} ->
        X*X;
      {rectangle,X,Y} ->
        X*Y
    end,
    area().
```

University of Kent

```
Pid = spawn(demo1, area, []),
Pid ! {square, 10}
```

```
-module(demo1).
-export([area/0]).

area() ->
    receive
      {square,X} ->
        X*X;
      {rectangle,X,Y} ->
        X*Y
    end,
    area().
```

University of Kent

```
area() ->
    receive
      {square,X} ->
        print(X*X);
      {rectangle,X,Y} ->
        print(X*Y)
    end,
    area().
```

Return to sender

```
Pid = spawn(Mod, area, []),
Pid ! {self(), {square, 10}},
receive
   Reply ->
      Reply
end.
```

```
-module(demo2).
-export([area/0]).

area() ->
   receive
     {From, {square,X}} ->
       From ! X*X;
     {From, {rectangle,X,Y}} ->
       From ! X*Y
   end,
   area().
```

Return to sender

```
Pid = spawn(Mod, area, []),
Pid ! {self(), {square, 10}},
receive
   {Pid, Reply} ->
      Reply
end.
```

```
-module(demo2).
-export([area/0]).

area() ->
   receive
     {From, {square,X}} ->
       From ! {self(), X*X};
     {From, {rectangle,X,Y}} ->
       From ! {self(), X*Y}
   end,
   area().
```

University of Kent

Four concurrency primitives

- `Pid = spawn(…)`
- `Pid ! Message`
- `receive Pattern -> Actions; … end`
- `self()`

```
receive
    Pattern1 ->
      Actions1;
    …
    PatternN ->
      ActionsN
after Time ->
    TimeoutActions
end
```

University of Kent

Registered processes

Only the parent knows its child

```
Pid = spawn(…)

Pid = spawn()
register(Name, Pid)

Pid = whereis(Name)
```