Concurrency

Johan Montelius

KTH

VT23

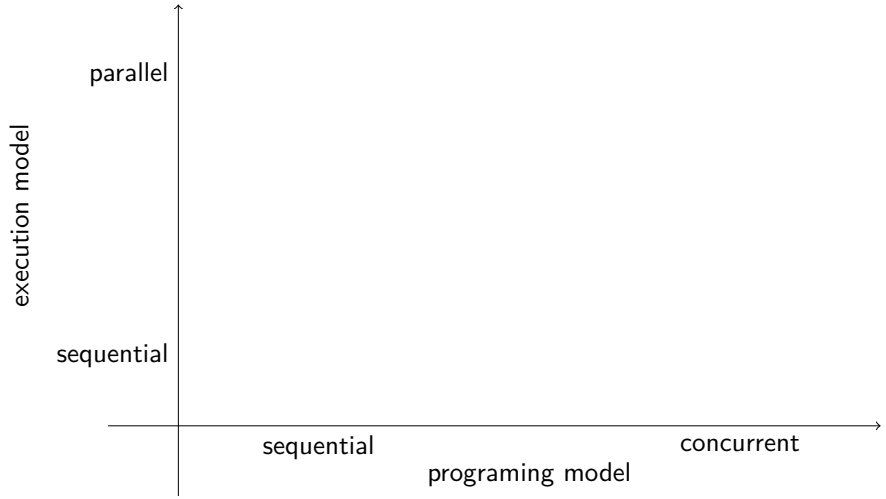
What is concurrency?

Concurrency: (the illusion of) happening at the same time.

A property of the programming model.

Why would we want to do things concurrently?

concurrency vs parallelism



concurrency models

- Shared memory: modify a shared data structure
  - C++/C
  - Java
- Message passing: processes send and receive messages
  - Erlang/Elixir
  - Go
  - Scala
  - Occam
  - Rust
  - Smalltalk

There are more, but these are the two large groups.

how do we send messages

- Communicating Sequential Processes (CSP), messages are sent **through channels**, a process can choose to read a message from one or more channels
  - Go, Occam, Rust
- Actor model, messages are sent **to a process**, a process reads implicitly from its own channel
  - Smalltalk, Erlang/Elixir, Scala

actor model

An actor:

- state: keeps a private state that can only be changed by the actor
- receive: has one channel of incoming *messages*
- execute: given a state and a received message, the actor can
  - send: send a number of messages to other actors
  - spawn: create a number of new actors
  - transform: modify its state an continue, or terminate

ordering of messages

Is the set of messages to an actor ordered?

In what order should the messages be handled?

The evaluation of a function is deterministic, how about the execution of an actor?

naming of actors

How can an actor direct a message to a specific actor?

Do we have a global naming scheme?

How do we find the identifier of an actor?

exceptions

What should we do if we're sending a message to an actor that has terminated?

What if we're waiting for a message that will never be sent?

Are sent messages guaranteed to arrive?

process identifier

We introduce one additional data structure:

$$\text{Structures} = \{\text{Process identifiers}\} \cup \text{Atoms} \cup \{\{s_1, s_2\} | s_i \in \text{Structures}\}$$

There is no term, nor pattern, that corresponds to an identifier.

spawn

A new process is spawned by giving it a function to evaluate,

... the result is a process identifier (pid).

```
pid = spawn(fn() -> ... end)
```

or

```
pid = spawn(module, name, [arg, ...])
```

In the later case, the function must be exported from the module.

send

Given a process identifier, an arbitrary data structure can be sent to the process.

```
send(pid, message)
```

In Erlang this was written using an operator `!`*, often called "bang".*

receive

We extend expressions:

```
<expression> ::=  <receive expression> | ...
```

```
<receive  expression> ::=
          receive do <clauses> end
```

similar to case expressions

example

```
def server(sum) do
  receive do
    {:add, x} ->
          server(sum + x)
    {:sub, x} ->
          server(sum - x)
  end
end
```


side effects

In a *pure functional* program, the only effect of evaluating an expressions is the returned value - not any more.

```
   :
bar(pid, 42)
bar(pid, 32)
   :
```

```
   :
x = foo(2)
y = foo(2)
   :
```

```
def bar(pid, msg) do
    send(pid,{:hello, msg})
end
```

```
def foo(x) do
   receive do
     {:hello, msg} -> msg + x
   end
end
```


Who am I?

One more built-in function:

```
myPid = self()
```

few extensions

Few extensions to the functional subset:

- pid: a process identifier as a data structure
- spawn: creating a process, returning a pid
- send: sending of messages to a pid
- receive: selective receive of messages
- self: the process identifier of the current process

All constructs can, apart from the receive statement, almost be given a functional interpretation.

Our operational semantics does not give us any understanding of the execution.

order of messages

Message passing is: unreliable FIFO.

```
   :
send(pid, {:this, :is, :message, 1})
send(pid, {:this, :is, :message, 2})
send(pid, {:this, :is, :message, 3})
   :
```

What could be the result at the receiving end?

How many messages are lost in reality?

order of messages

Process one:

```
   :
send(pid, {:one, 1})
send(pid, {:one, 2})
   :
```

Process two:

```
   :
send(pid, {:two, 1})
send(pid, {:two, 2})
   :
```


selective receive

```
def sum(s) do
  receive do
    {:add, x} -> sum(s + x)
    {:sub, x} -> sum(s - x)
    {:mul, x} -> sum(s * x)
  end
end
```

Assume we spawn a process given the expression `fn() -> sum(10) end`, and the sequence of messages in the queue is:

```
{:sub, 4}, {:add, 10}, {:mul, 4}, {:mul, 2}, {:sub, 10}
```

selective receive

```
def closed(s) do
  receive
    {:add, x} -> closed(s + x)
    :open -> open(s)
    :done -> {:ok, s}
  end
end
```

```
def open(s) do
  receive do
    {:mul, x} -> open(s * x)
    {:sub, x} -> open(s - x)
    :close -> closed(s)
  end
end
```

Assume we spawn `fn() -> closed(4) end` and the sequence of messages is:

`{:sub, 4}, :open, {:mul, 4}, {:add, 2}, :close {:add, 2}, :done`

In every receive expression we start from the beginning of the queue.

implicit deferral

Selective receive: we specify which messages we are willing to accept.

Implicit deferral: messages that we do not explicitly receive, remain in the message queue.

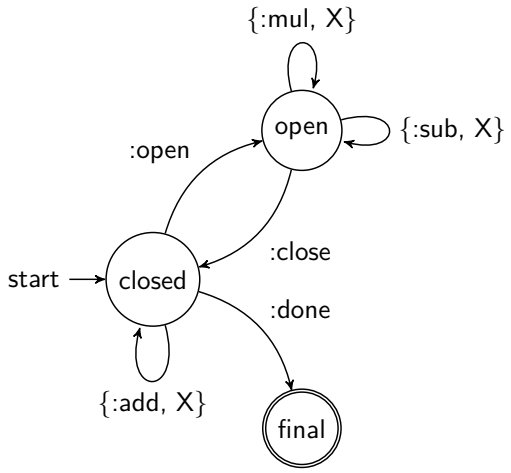
We could have chosen *fifo receive* i.e. messages must be received in the order they have in the message queue (Actors model).

We could have chosen *explicit deferral*, but then we would have to state which messages that should be handled later.

how to describe a processes

- Finite state machine (FSM) : describing the states where messages determine transitions
- Sequence diagram : how processes interact, protocol definitions
- Flow-based Programming (FBP) : architecture view of processes
- Domain Specific Language : describe the systems in a high level programming language
- ...

finite state machine - FSM



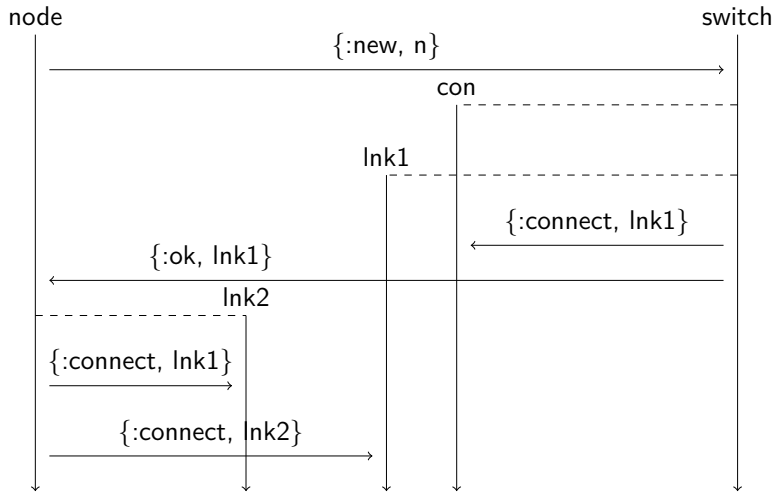
finite state machine - FSM

Elixir receive statements are not a direct realization of a finite state machine.

Messages that arrive too early in a finite state automata would give us an undefined state.

The *implicit deferral* give us a very simple description of a finite state machine where messages are allowed to arrive too early.

sequence diagram



flow-based programming

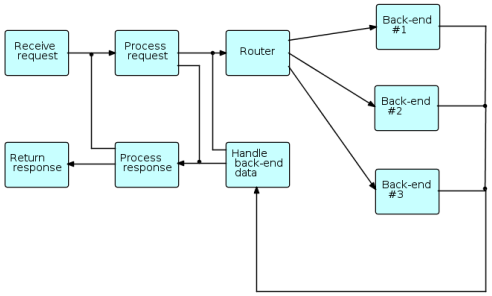


Figure: from J Paul Morrison www.jpaulmorrison.com

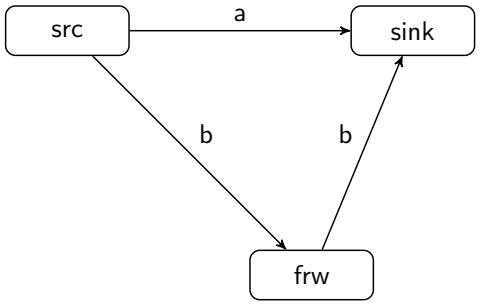
time to deliver

src/2

```
def src(sink, frw) do
    send(sink, :a)
    send(frw, :b)
end
```

frw/2

```
def frw(sink) do
   receive do
     msg -> send(sink,msg)
   end
end
```



example account

```
def acc(saldo) do
  recieve do
     {:deposit, money} ->
        acc(saldo + money)
     {:withdraw, money} ->
        acc(saldo - money)
  end
end
```

```
def doit() do
  acc = spawn(fn()->acc(0) end)
  send(acc, {:deposit, 20})
  send(acc, {:withdraw, 10})
  acc
end
```


example account

```
def acc(saldo) do
  recieve do
    {:deposit, money} ->
       acc(saldo + money)
    {:withdraw, money} ->
       acc(saldo - money)
    {:request, from} ->
       send(from, {:saldo, saldo})
       acc(saldo)
  end
end
```

```
def check(acc) do
  send(acc, {:request, self()})
  receive do
    {:saldo, saldo} ->
        saldo
  end
end
```


summary

- asynchronous: messages are sent and eventually (hopefully) delivered
- FIFO: message delivery is ordered
- selective receive: the receiver decides the order of handling messages
- implicit deferral: messages remain in the queue until handled
- diagrams: finite state machines, sequence diagrams, flow-based program