Concurrency control abstractions (PDCS 9, CPE 5*)

Carlos Varela
Rennselaer Polytechnic Institute

October 15, 2021

* Concurrent Programming in Erlang, by J. Armstrong, R. Virding, C. Wikström, M. Williams

Actor Languages Summary

- Actors are concurrent entities that react to messages.
  - State is completely encapsulated. There is no shared memory!
  - Message passing is asynchronous.
  - Actors can create new actors. Run-time has to ensure fairness.
- AMST extends the call by value lambda calculus with actor primitives. State is modeled as function arguments. Actors use `ready` to receive new messages.
- Erlang extends a functional programming language core with processes that run arbitrary functions. State is implicit in the function's arguments. Control loop is explicit: actors use `receive` to get a message, and tail-form recursive call to continue. Ending a function denotes process (actor) termination.
- SALSA extends an object-oriented programming language (Java) with universal actors. State is explicit, encapsulated in instance variables. Control loop is implicit: ending a message handler, signals readiness to receive a new message. Actors are garbage-collected.

Causal order

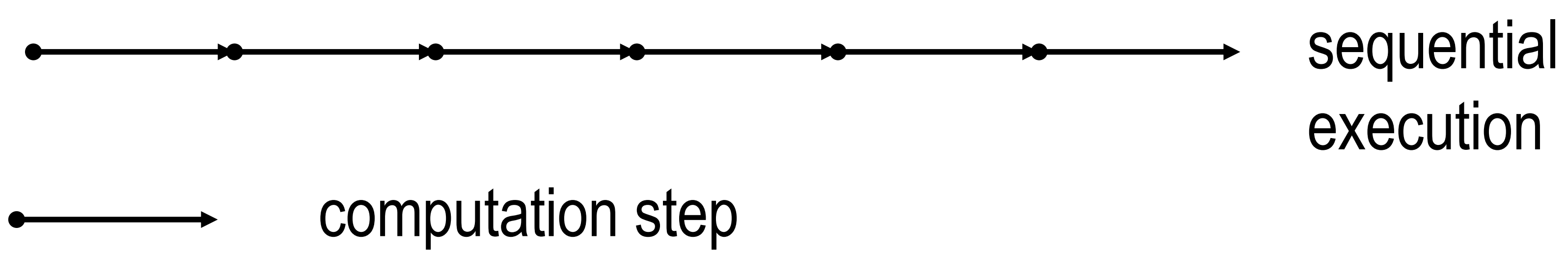
- In a sequential program all execution states are totally ordered

- In a concurrent program all execution states of a given actor are totally ordered

- The execution state of the concurrent program as a whole is partially ordered

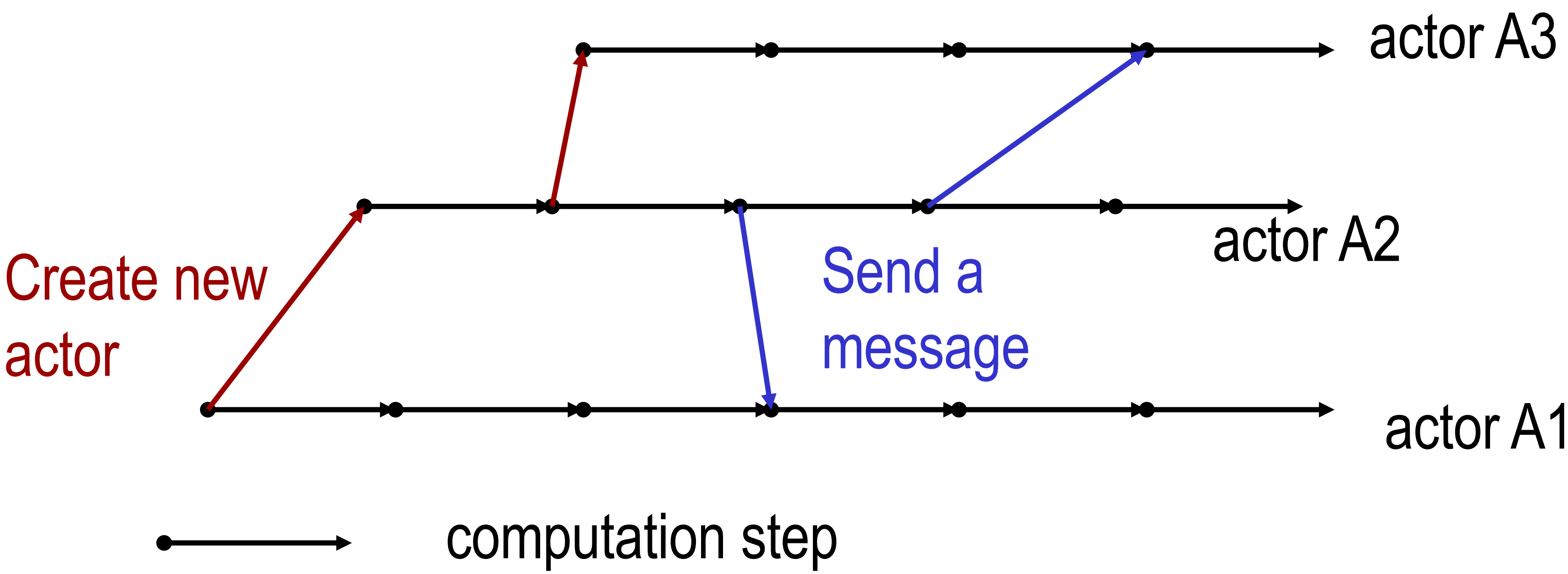
Total order

- In a sequential program all execution states are totally ordered



Causal order in the actor model

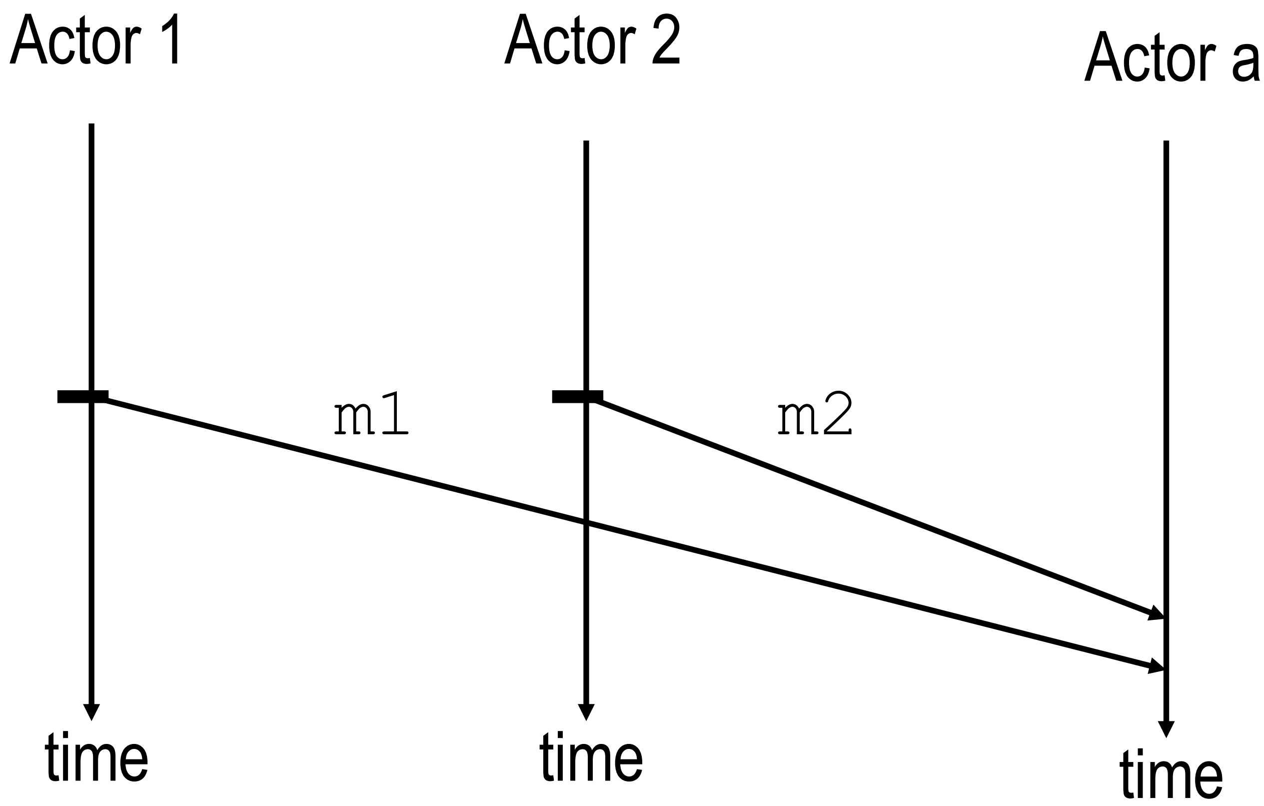
- In a concurrent program all execution states of a given actor are totally ordered
- The execution state of the concurrent program is partially ordered



Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is asynchronous message passing
  - Messages can arrive or be processed in an order different from the sending order.

Example of nondeterminism



Actor `a` can receive messages `m1` and `m2` in any order.

Tree Product Behavior Revisited

Notice we use token-passing continuations (`@`,`token`), a join block (`join`), and a first-class continuation (`currentContinuation`).

```
module treeprod;
import tree.Tree;

behavior JoinTreeProduct {

    int multiply(Object[] results){
      return (Integer) results[0] * (Integer) results[1];
    }
    int compute(Tree t){
      if (t.isLeaf()) return t.value();
      else {
        JoinTreeProduct lp = new JoinTreeProduct();
        JoinTreeProduct rp = new JoinTreeProduct();
        join {
          lp <- compute(t.left());
          rp <- compute(t.right());
        } @ multiply(token) @ currentContinuation;
      }
    }
}
```

Concurrency Control in SALSA

- SALSA provides three main coordination constructs:
  - Token-passing continuations
    - To synchronize concurrent activities
    - To notify completion of message processing
    - Named tokens enable arbitrary synchronization (data-flow)
  - Join blocks
    - Used for barrier synchronization for multiple concurrent activities
    - To obtain results from otherwise independent concurrent processes
  - First-class continuations
    - To delegate producing a result to another message, or actor

Token Passing Continuations

- Ensures that each message in the continuation expression is sent after the previous message has been **processed**. It also enables the use of a message handler return value as an argument for a later message (through the token keyword).

  – Example:

```
a1 <- m1() @
a2 <- m2( token );
```

Send `m1` to `a1` asking `a1` to forward the result of processing `m1` to `a2` (as the argument of message `m2`).

Token Passing Continuations

- `@` syntax using `token` as an argument is syntactic sugar.
  - Example 1:

```
a1 <- m1() @
a2 <- m2( token );
```

is syntactic sugar for:

```
token t = a1 <- m1();
a2 <- m2( t );
```

  - Example 2:

```
a1 <- m1() @
a2 <- m2();
```

is syntactic sugar for:

```
token t = a1 <- m1();
a2 <- m2():waitfor( t );
```

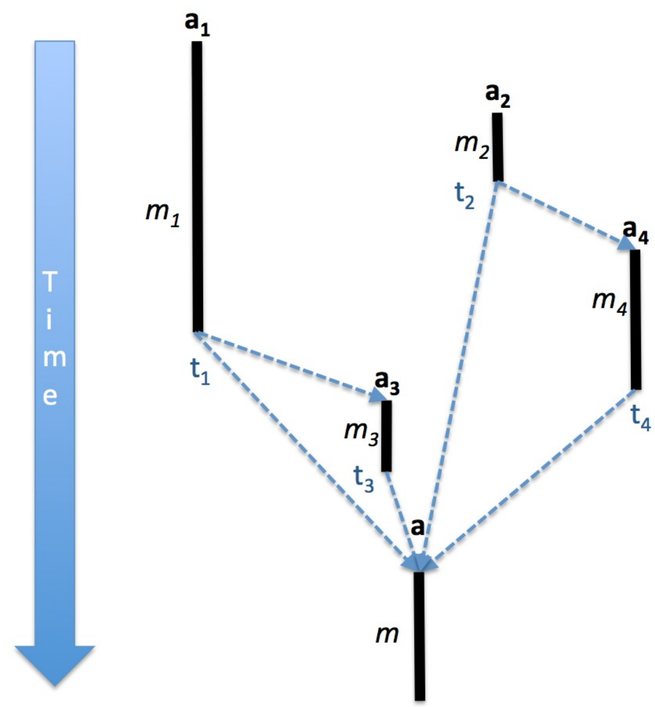
Named Tokens

- Tokens can be named to enable more loosely-coupled synchronization

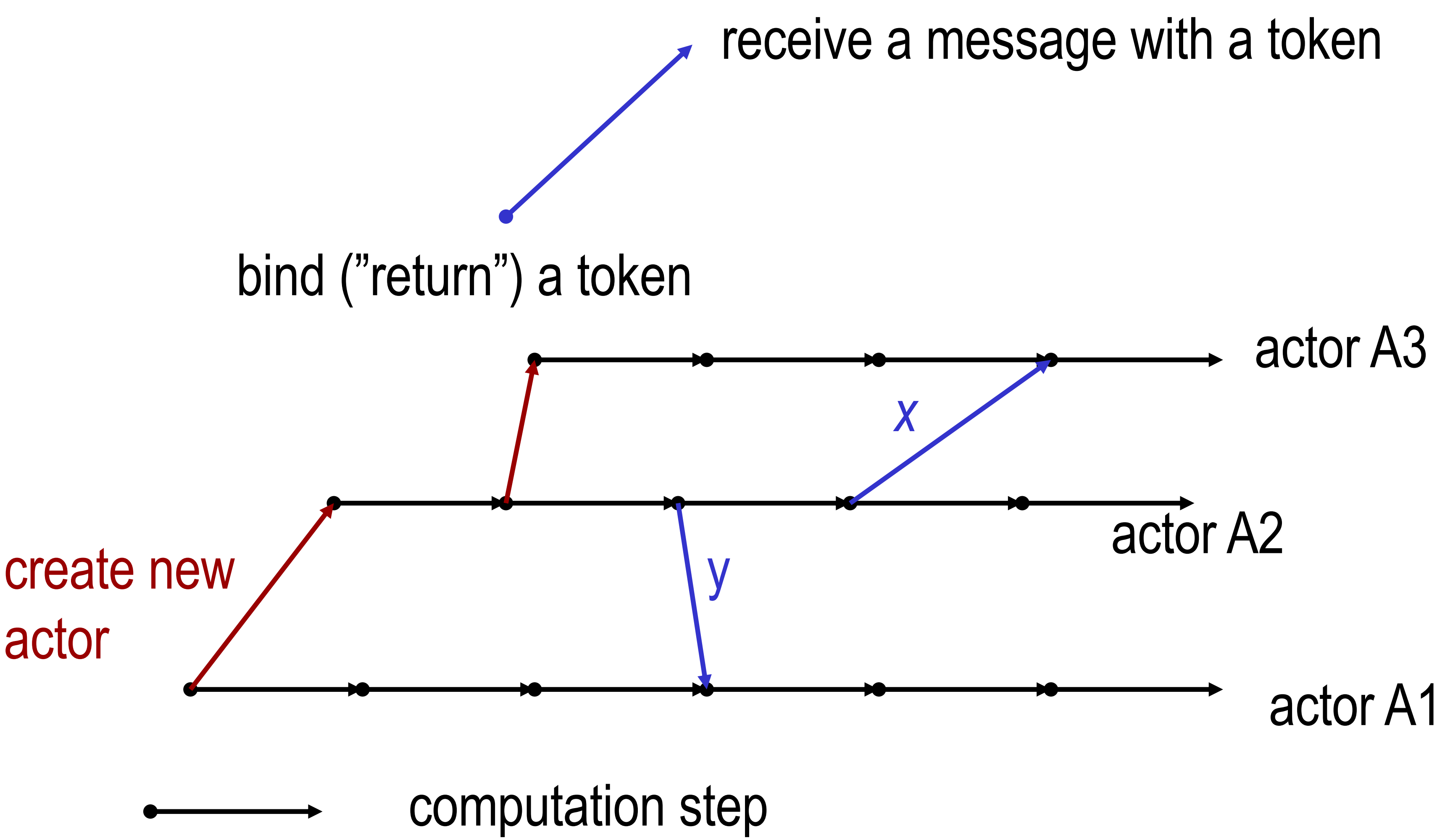
  – Example:

```
token t1 = a1 <- m1();
token t2 = a2 <- m2();
token t3 = a3 <- m3( t1 );
token t4 = a4 <- m4( t2 );
a <- m(t1,t2,t3,t4);
```

Sending `m(...)` to `a` will be delayed until messages `m1()..m4()` have been processed. `m1()` can proceed concurrently with `m2()`.



Causal order in the actor model



Deterministic Cell Tester Example

```
module cell;

behavior TokenCellTester {

    void act( String[] args ) {

        Cell c = new Cell(0);
        standardOutput <- print( "Initial Value:" ) @
        c <- get() @
        standardOutput <- println( token ) @
        c <- set(2) @
        standardOutput <- print( "New Value:" ) @
        c <- get() @
        standardOutput <- println( token );

    }
}
```

`@` syntax enforces a sequential order of message execution.

`token` can be optionally used to get the return value (completion proof) of the previous message.

Cell Tester Example with Named Tokens

```
module cell;

behavior NamedTokenCellTester {

    void act(String args[]){

          Cell c = new Cell(0);
          token p0 = standardOutput <- print("Initial Value:");
          token t0 = c <- get();
          token p1 = standardOutput <- println(t0):waitfor(p0);
          token t1 = c <- set(2):waitfor(t0);
          token p2 = standardOutput <- print("New Value:"):waitfor(p1);
          token t2 = c <- get():waitfor(t1);
          standardOutput <- println(t2):waitfor(p2);
    }
}
```

We use `p0, p1, p2` tokens to ensure printing in order.

We use `t0, t1, t2` tokens to ensure cell messages are processed in order.

Join Blocks

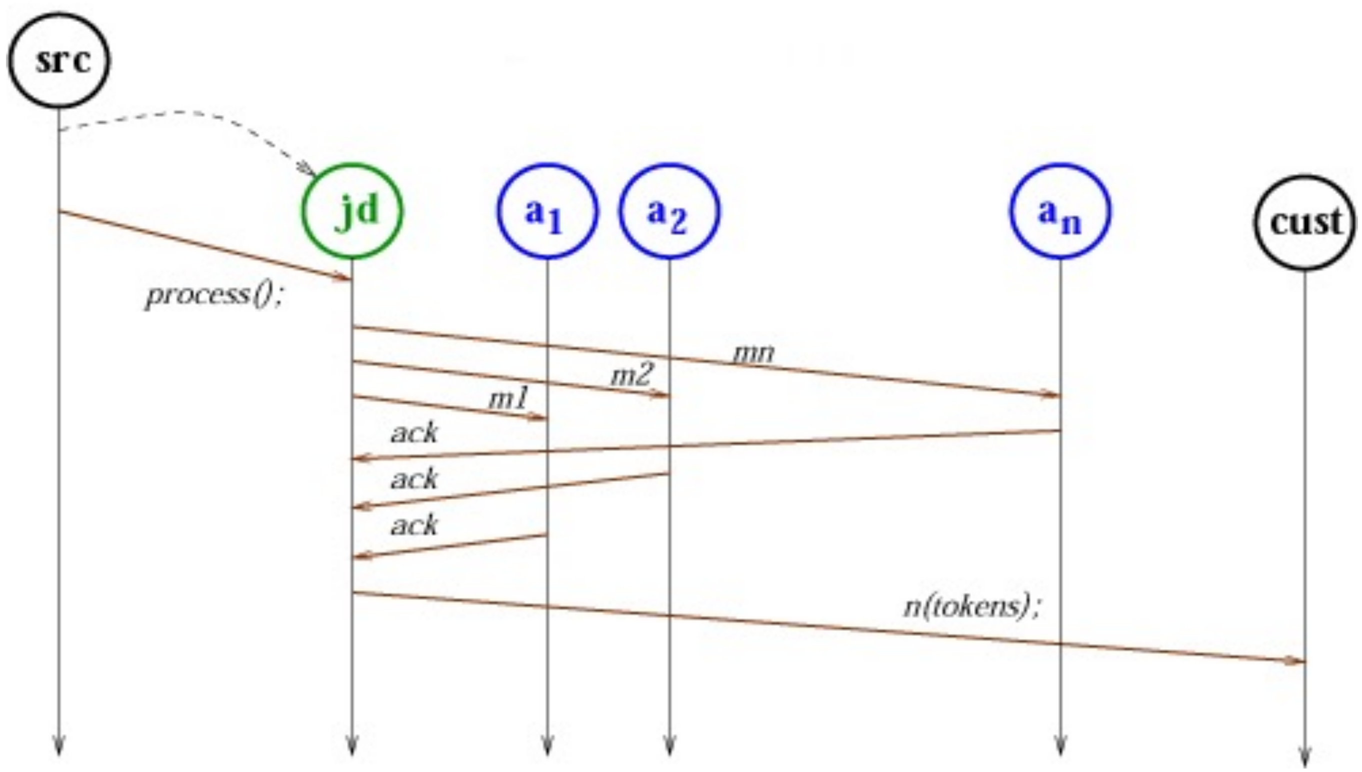
- Provide a mechanism for synchronizing the processing of a set of messages.
- Set of results is sent along as a *token* containing an array of results.
  - Example:

```
UniversalActor[] actors = { searcher0, searcher1,
                            searcher2, searcher3 };
join {
  for (int i=0; i < actors.length; i++){
     actors[i] <- find( phrase );
  }
} @ resultActor <- output( token );
```

Send the `find( phrase )` *message to each actor in* `actors[]` *then after all have completed send the result to* `resultActor` *as the argument of an* `output( … )` *message.*

Example: Acknowledged Multicast

```
join{ a1 <- m1(); a2 <- m2(); … an <- mn(); } @
   cust <- n(token);
```



Lines of Code Comparison

	Java	Foundry	SALSA
Acknowledged Multicast	168	100	31

First Class Continuations

- Enable actors to delegate computation to a third party independently of the processing context.

- For example:

```
int m(…){
    b <- n(…) @ currentContinuation;
}
```

Ask (delegate) actor `b` to respond to this message `m` on behalf of current actor (`self`) by processing `b`'s message `n`.

Delegate Example

```
fib(15)
```

is syntactic sugar for:

```
self <- fib(15)
```

```
module fibonacci;

behavior Calculator {

   int fib(int n) {
        Fibonacci f = new Fibonacci(n);
        f <- compute() @ currentContinuation;
   }
   int add(int n1, int n2) {return n1+n2;}

   void act(String args[]) {
        fib(15) @ standardOutput <- println(token);
        fib(5) @ add(token,3) @
        standardOutput <- println(token);
   }
}
```

Fibonacci Example

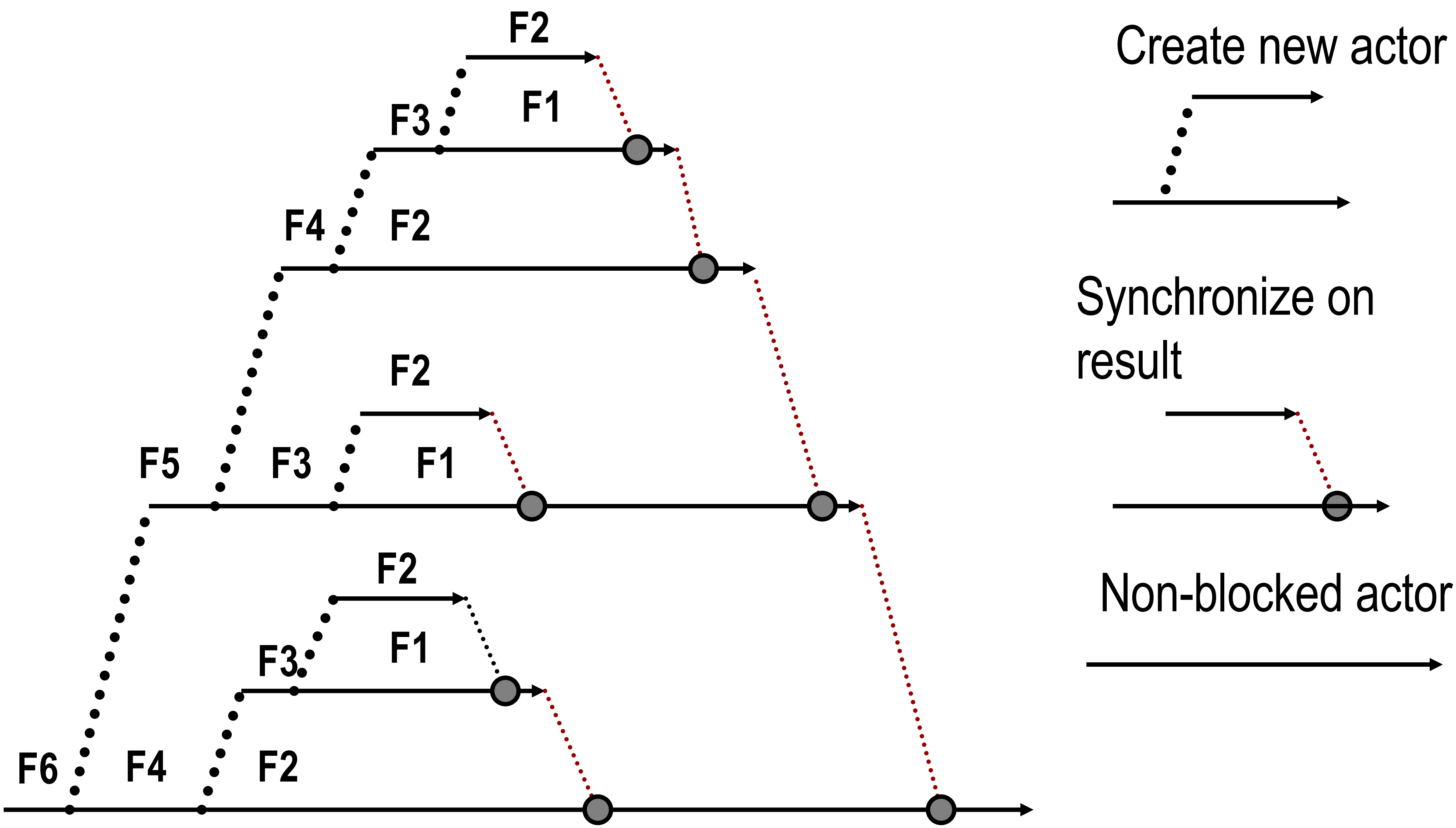
```
module fibonacci;

behavior Fibonacci {
    int n;

    Fibonacci(int n)           { this.n = n; }

    int add(int x, int y) { return x + y; }

    int compute() {
          if (n == 0)          return 0;
          else if (n <= 2)     return 1;
          else {
                    Fibonacci fib1 = new Fibonacci(n-1);
                    Fibonacci fib2 = new Fibonacci(n-2);
                    token x = fib1<-compute();
                    token y = fib2<-compute();
                    add(x,y) @ currentContinuation;
          }
    }

    void act(String args[]) {
          n = Integer.parseInt(args[0]);
          compute() @ standardOutput<-println(token);
    }
}
```

Fibonacci Example 2

```
module fibonacci2;

behavior Fibonacci {

    int add(int x, int y) { return x + y; }

    int compute(int n) {
        if (n == 0)      return 0;
        else if (n <= 2) return 1;
        else {
                Fibonacci fib = new Fibonacci();
                token x = fib <- compute(n-1);
                compute(n-2) @ add(x,token) @ currentContinuation;
        }
    }

    void act(String args[]) {
        int n = Integer.parseInt(args[0]);
        compute(n) @ standardOutput<-println(token);
    }
}
```

`compute(n-2)` is a message to `self`.

Execution of `salsa Fibonacci 6`



Concurrency control in Erlang

- Erlang uses a *selective receive* mechanism to help coordinate concurrent activities:
  - Message patterns and guards
    - To select the next message (from possibly many) to execute.
    - To receive messages from a specific process (actor).
    - To receive messages of a specific kind (pattern).
  - Timeouts
    - To enable default activities to fire in the absence of messages (following certain patterns).
    - To create timers.
  - Zero timeouts (`after 0`)
    - To implement priority messages, to flush a mailbox.

Selective Receive

```
receive
    MessagePattern1 [when Guard1] ->
       Actions1 ;
    MessagePattern2 [when Guard2] ->
       Actions2 ;
    …
end
```

`receive` suspends until a message in the actor's mailbox matches any of the patterns including optional guards.

- Patterns are tried in order. On a match, the message is removed from the mailbox and the corresponding pattern's actions are executed.
- When a message does not match any of the patterns, it is left in the mailbox for future `receive` actions.

Selective Receive Example

Example program and mailbox (head at top):

```
receive
    msg_b -> …
end
```

msg_a
msg_b
msg_c

`receive` tries to match `msg_a` and fails. `msg_b` can be matched, so it is processed. Suppose execution continues:

```
receive
    msg_c -> …
    msg_a -> …
end
```

msg_a
msg_c

The next message to be processed is `msg_a` since it is the next in the mailbox and it matches the 2nd pattern.

Receiving from a specific actor

```
Actor ! {self(), message}
```

`self()` is a Built-In-Function (BIF) that returns the current (executing) process id (actor name). Ids can be part of a message.

```
receive
    {ActorName, Msg} when ActorName == A1 ->
        …
end
```

`receive` can then select only messages that come from a specific actor, in this example, `A1`. (Or other actors that know A1's actor name.)

Receiving a specific kind of message

```
counter(Val) ->
  receive
    increment -> counter(Val+1);
    {From,get} ->
      From ! {self(), Val},
      counter(Val);
    stop -> true;
    Other -> counter(Val)
  end.
```

`increment` is an atom whereas `Other` is a variable (that matches anything!).

`counter` is a behavior that can receive `increment` messages, `get` request messages, and `stop` messages. `Other` message kinds are ignored.

Order of message patterns matters

```
receive
    {{Left, Right}, Customer} ->
       NewCust = spawn(treeprod,join,[Customer]),
       LP = spawn(treeprod,treeprod,[]),
       RP = spawn(treeprod,treeprod,[]),
       LP!{Left,NewCust},
       RP!{Right,NewCust};
    {Number, Customer} ->
       Customer ! Number
end
```

`{Left,Right}` is a more specific pattern than `Number` is (which matches anything!). Order of patterns is important.

In this example, a binary tree is represented as a tuple `{Left, Right}`, or as a `Number`, e.g.,

`{{{5,6},2},{3,4}}`

Selective Receive with Timeout

```
receive
    MessagePattern1 [when Guard1] ->
        Actions1 ;
    MessagePattern2 [when Guard2] ->
        Actions2 ;
    …
    after TimeOutExpr ->
        ActionsT
end
```

`TimeOutExpr` evaluates to an integer interpreted as *milliseconds*.

If no message has been selected within this time, the timeout occurs and `ActionsT` are scheduled for evaluation.

A timeout of `infinity` means to wait indefinitely.

Timer Example

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

`sleep(Time)` suspends the current actor for `Time` milliseconds.

Timeout Example

```
receive
    click ->
       receive
           click ->
             double_click
       after double_click_interval() ->
             single_click
       end
    ...
end
```

`double_click_interval` evaluates to the number of milliseconds expected between two consecutive mouse clicks, for the receive to return a `double_click`. Otherwise, a `single_click` is returned.

Zero Timeout

```
receive
    MessagePattern1 [when Guard1] ->
       Actions1 ;
    MessagePattern2 [when Guard2] ->
       Actions2 ;
    …
    after 0 ->
       ActionsT
end
```

A timeout of `0` means that the timeout will occur immediately, but Erlang tries all messages currently in the mailbox first.

Zero Timeout Example

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

`flush_buffer()` completely empties the mailbox of the current actor.

Priority Messages

```
priority_receive() ->
       receive
          interrupt ->
             interrupt
          after 0 ->
             receive
                AnyMessage ->
                   AnyMessage
             end
       end.
```

`priority_receive()` will return the first message in the actor's mailbox, except if there is an `interrupt` message, in which case, `interrupt` will be given priority.

Exercises

46. Download and execute the reference cell and tree product examples in SALSA and Erlang.
47. Write a solution to the Flavius Josephus problem in SALSA and Erlang. A description of the problem is at CTM Section 7.8.3 (page 558).
48. PDCS Exercise 9.6.6 (page 204).
49. How would you implement token-passing continuations, join blocks, and first-class continuations in Erlang?
50. How would you implement selective receive in SALSA?