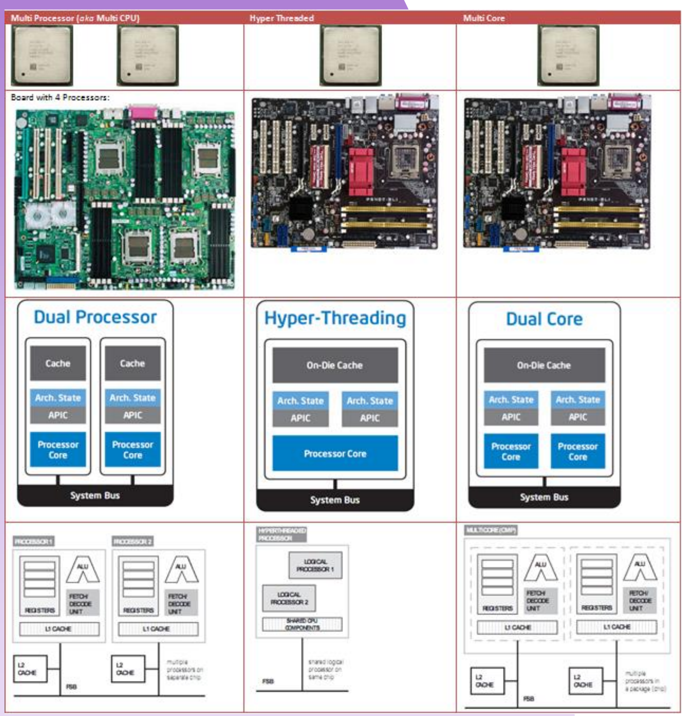
Programming languages

Concurrent programming in Erlang

Concurrent computing

- A form of computation in which several procedures are executed during overlapping periods of time instead of being executed sequentially.
- The execution can be carried out on one physical processor (CPU) or on several.

Parallel computing hardware



- Form of computation in which several procedures are executed simultaneously in:
  - **Multiprocessor** (**SMP**)– multiple physical processors (CPUs) connected by memory or network.
  - **Multicore** (**CMP**)– multiple physical processors within the same chip.
  - **Multithreaded** (**SMT**)– simulates multiple logical processors within a single physical processor.

Processes and threads

- A **process** is an instance of a running program
  - There may be several processes executing the same program, but each one is a different process, with its own representation (*PCBs*)
- A **thread** of execution is the smallest sequence of programmed instructions that can be handled independently by a scheduler.
  - The scheduler is the one who decides how to give access to the resources of a system (processor time, communication bandwidth, etc.)

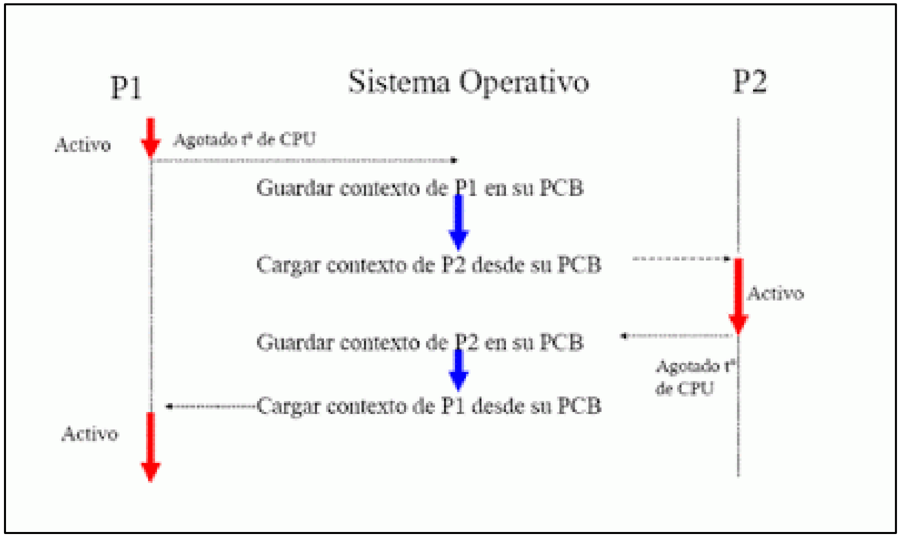
Processes

- A process consists of at least:
  - The program code.
  - The data of the program.
  - An execution stack.
  - The PC (program counter) indicating the next instruction to be executed.
  - A general-purpose register set with current values.
  - A set of OS resources (memory, open files, etc.)

For CPU scheduling it is the processes that are important, not the programs.

Concurrent processes

- They can be executed in a single core by interleaving the execution steps of each process through time slices (**preemptive multitask**).
- Only one process is executed at a time, and if it does not complete during the time segment, it is paused, and another process starts or resumes its execution, and then resumes the original process.



Context switch

- When a process is running, its PC, stack pointer, registers, etc., are loaded into the CPU (i.e., the hardware registers contain the current values).
- When a running process is stopped, it saves the current values of these registers (**context**) on the PCB of that process.
- The action of switching the CPU from one process to another is called **context switch**.
- Timesharing systems perform 100 to 1,000 context switches per second.
- This job is **overload**.

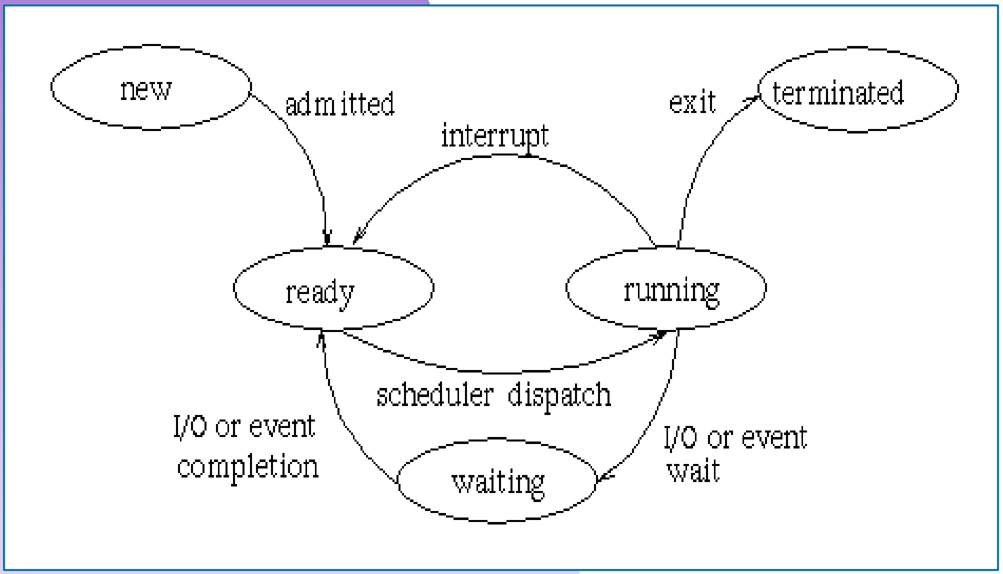
PCB (Process Control Block)

- Data structure that represents the process, that is, it contains the information associated with each process:
  - Current state of the process.
  - CPU register values.
  - Planning information.
  - Information for memory management.
  - I/O status information.
  - Accounting or statistical information.
  - Event by which the process is blocked.

PCB's and status queues

- The concurrent computing system maintains a collection of queues that represent the state of all processes in the system.
  - There is typically one queue per state.
  - Each PCB is in a status queue according to its current status.
  - As a process changes state, its PCB is removed from one queue and added to another.

States and transitions



- Every process has a running state that indicates what it is currently doing:
  - **New** – The process is being created.
  - **Running** - executing instructions in the CPU.
  - **Ready** - CPU standby.
  - **Waiting** - Waiting for an event.
  - **Terminated** – The process finished its execution.
- During its life in the system, a process goes from one state to another.

Programming model

- Describes the **form of interaction and concurrent communication.**
- In some concurrent computing systems, it has been hidden from the programmer.
- In others, it must be handled explicitly.
- Explicit communication can be divided into 2 classes:
  - **Shared memory**
  - **Message passing**

Shared memory

- Concurrent components communicate by altering the content of shared memory locations *(Java or C#).*

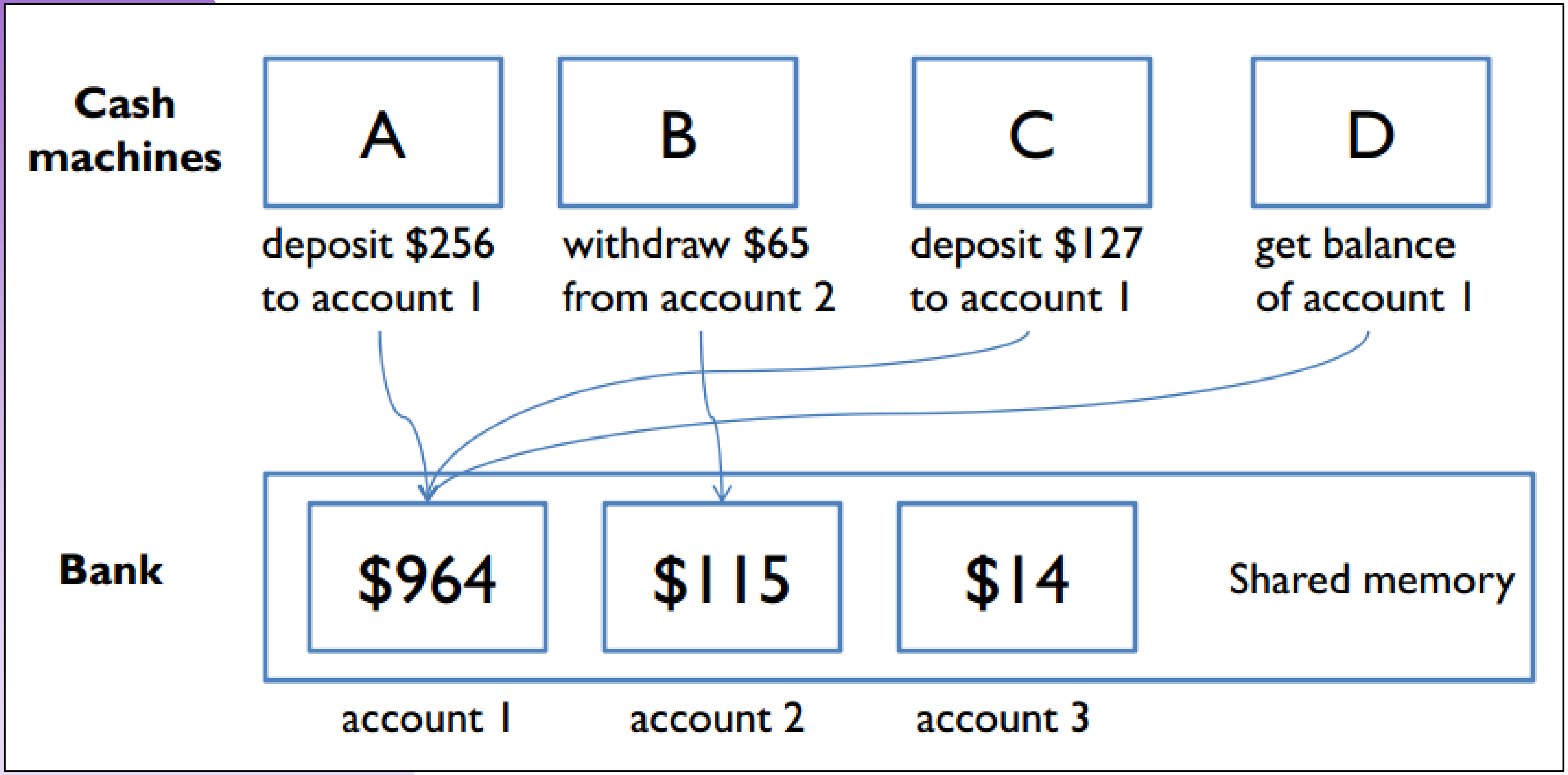
- This style of concurrent programming usually requires some form of locking to be applied (**mutexes**, **semaphores,** or **monitors**) for coordination (**synchronization**) between processes or threads.

- A program that properly implements any of these mechanisms is said to be thread-safe (**thread-safe**).

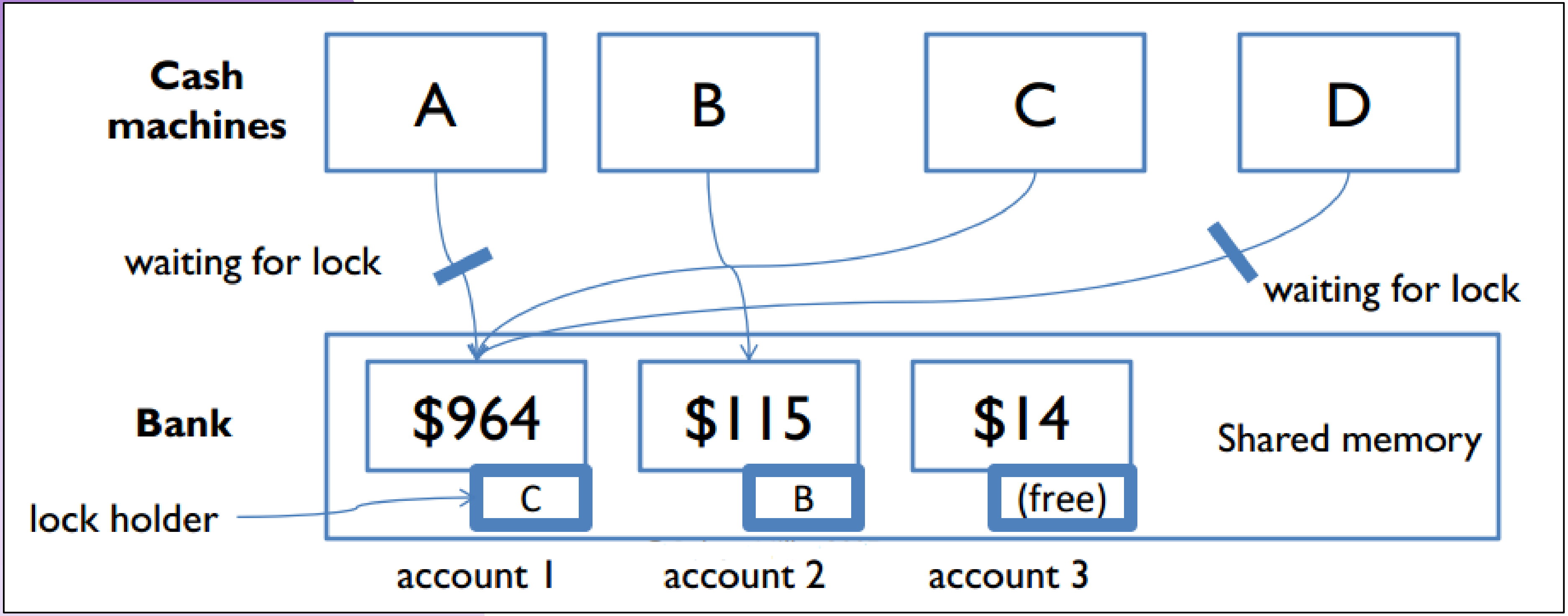
Dangers of concurrent execution

- **Race condition:** When several processes access at the same time and change the state of a shared resource (for example, a variable), thus obtaining a value that depends on the order of their execution.
- If they are not synced correctly, data corruption may occur.
- **Example**: Concurrent access via ATMs from several clients to shared bank accounts.

Example: Use of ATMs



Possible solution: use of locks



More dangers...

- **Deadlock**: is the permanent blocking of a set of processes or threads of execution in a concurrent system that compete for system resources or communicate with each other.
- There is no general solution to deadlocks.
- **Example**: 2 children who want to shoot a bow, but one grabbed the bow and the other the arrow, and they wait for the other to drop what he grabbed.

More dangers...

- ***Livelock:*** is like a *deadlock*, except that the state of the two processes involved in the *livelock* constantly changes with respect to the other.
- **Example**: 2 people in a narrow hallway blocking each other and moving in unison to let the other through, maintaining the block.

More dangers...

- **Starvation**: when a process or a thread of execution is always denied access to a shared resource that it requires to finish its task.

- **Example**: Problem of the philosopher's dinner.

Concepts to consider

- **Mutual Exclusion (ME)** refers to the requirement to ensure that no two processes are in their critical section at the same time to prevent race conditions.

- A **critical section** is a piece of code that Access a shared resource (data structure or device) that should not be accessed concurrently by more than one thread of execution.

ME tools

- **Mutexes** are flags that are used to indicate when a resource can be used.

- A **semaphore** is a variable or abstract data type that records how many units of a particular resource are available, coupled with operations for safe tuning (no race conditions), and is used to control access to a common resource by several concurrent processes.

ME tools

- A **monitor** is a synchronization mechanism that allows threads to execute with mutual exclusion and have the ability to wait (block) until a certain condition becomes true.

- A monitor is made up of a **mutex** and **condition variables**.

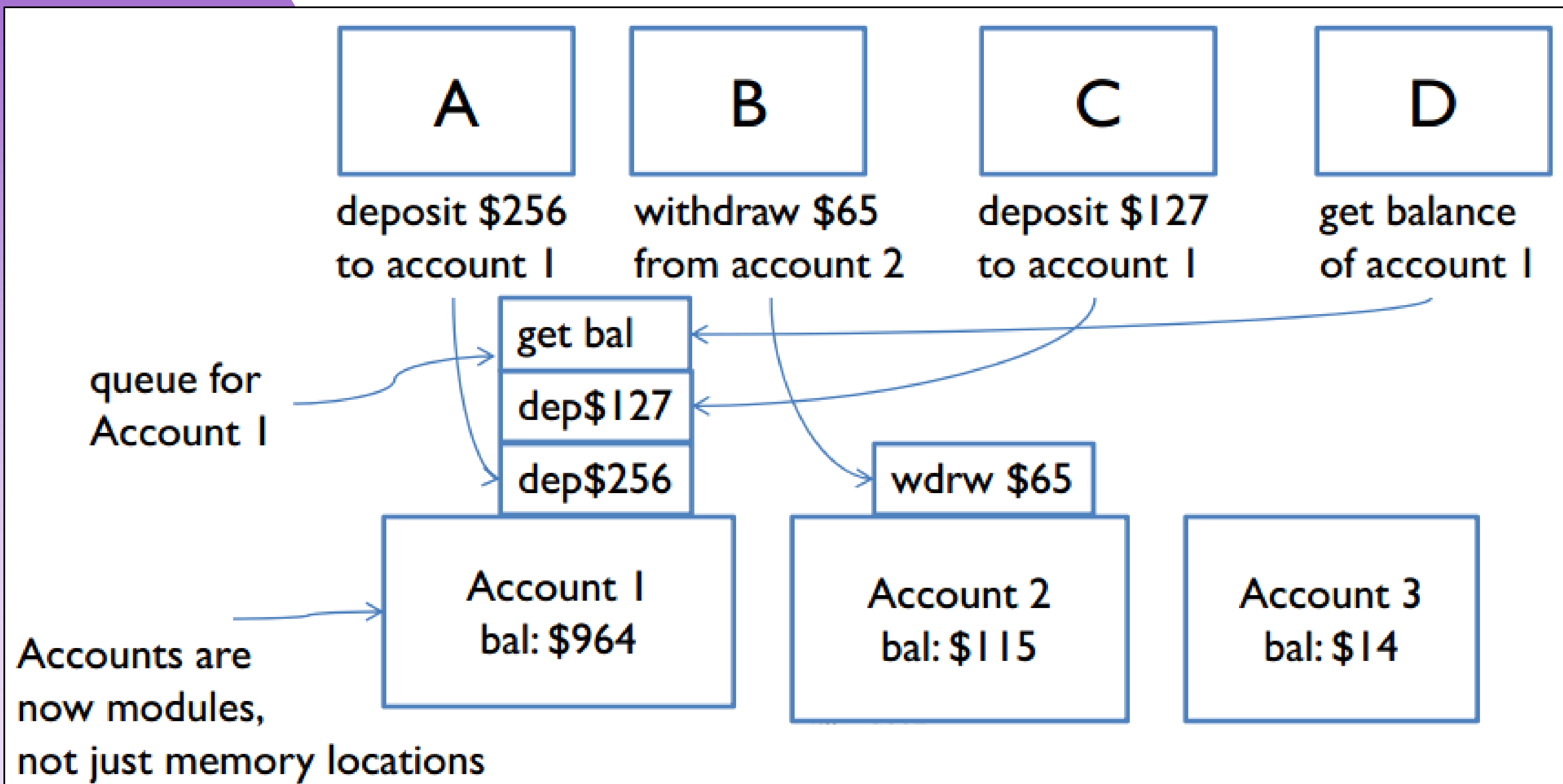
Message passing

- Concurrent components communicate by exchanging messages (*Erlang, Go, & Occam*).
- The exchange of messages can be done **asynchronously**, or you can use a **synchronous** style (*rendezvous)* in which the sender is blocked until the message is received.
- Asynchronous message passing can be reliable or unreliable (*send and pray*).
- This form of communication tends to be **easier to reason** than shared memory and is typically considered a **more robust** form of concurrent programming.

Message passing interaction

- Received messages (requests) are queued to be handled one at a time.
- The sender will not stop working while waiting for his message to be answered, so he will continue to attend messages in his own queue.
- Responses eventually come back through other messages.

Example: Use of ATMs



Dangers in message passing

- Does not remove race conditions.
  - **Example**: send money withdrawal messages without checking if there are enough funds.
- Nor does it remove *deadlocks*.
  - **Example**: Two processes are left waiting for responses to messages to respond to messages.

Concurrent language

- We will use a concurrent programming language based on the message passing model.
- Install the programming language Erlang http://www.erlang.org/

Processes and concurrence in Erlang

- In Erlang, concurrency is implemented by **creating and communicating processes**.
- **Process**: a separate, self-contained unit of computation that runs concurrently with other processes on the system.
- Erlang processes **do not share memory (data)** with other processes.
- Processes communicate through message passing (**concurrent programming model**)

Processes

- In Erlang, the processes belong to the programming language and not to the Operating System.
- In Erlang, programming with processes is easy, since you only need 3 primitives:
  - **spawn:** to create processes
  - **send**: to send messages, and
  - **receive:** to receive messages.

Process creation

- **spawn/1** or **spawn/3** create a new concurrent process and return its identifier.

```
Pid = spawn(Function)
Pid = spawn(Module, Function, ArgList)
```

- Process identifiers (**pid**) are used for message exchange.
- The call does **not wait** for the function to be evaluated (it returns immediately).
- The process **automatically terminate** when the function finishes executing.
- The **return value** of the process is lost.

Example

Write its argument N times

Create 2 concurrent processes

```
-module(talk).
-export([start/0, say_something/2]).

say_something(_, 0) ->
    done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(talk, say_something, [hello, 3]),
    spawn(talk, say_something, [bye, 3]).
```

Example

```
5> c(talk).
{ok, talk}

6> talk:say_something(hello, 3).
hello
hello
hello
done

7> talk:start().
hello
bye
hello
Bye
<0.44.0>
hello
bye
```

PID of the second process (latest)

Sending messages

- A message is sent to another process using the '!' (***send***) primitive, like so:

  `Pid ! Message`

- **`Pid`** is the identifier of the process to which the message is sent.
- Sending the message is **asynchronous**
  - The sender continues with what he was doing (**does not wait**).
  - The system does not inform the sender if the message was delivered, even if the destination process no longer exists.
  - The application must implement all forms of required checking.

Sending messages

- The **message** can be any valid Erlang term.
- The **return value of !** is the message it sends, so:

  `Pid1 ! Pid2 ! … ! Message`

- would send the same message to all processes **`Pid1`**, **`Pid2`**, …
- If the receiver has not finished, all messages are delivered to him in the same order in which they are sent.

Message reception

- the **receive** primitive is used to receive messages, with the following syntax:

```
receive
    Pattern1[when Guard1] ->
          Actions1;
    Pattern2[when Guard2] ->
          Actions2;
    ...
end
```

- Each process has its own **mailbox**
- All messages sent to a process are stored in its mailbox in the order they are received.

Example

```
-module(area_server).
-export([cycle/0]).

cycle() ->
   receive
      {rectangle, Width, Height} ->
         io:format("Area of rectangle = ~p~n",[Width * Height]),
         cycle();
      {circle, R} ->
         io:format("Area of circle = ~p~n", [3.14159 * R * R]),
         cycle();
      Other ->
         io:format("I don't know the area of ~p~n" ,[Other]),
         cycle()
 end.
```

Example

```
1> Pid = spawn(fun area_server:cycle/0).
<0.36.0>
2> Pid ! {rectangle, 6, 10}.
Area of rectangle = 60
{rectangle, 6, 10}
3> Pid ! {circle, 23}.
Area of circle = 1661.90
{circle, 23}
4> Pid ! {triangle,2,4,5}.
I don't know the area of {triangle,2,4,5}
{triangle,2,4,5}
```

Message reception

- When a message is received, the system tries to **match it sequentially** with some of the patterns (and with their possible guards).
  - If a **message match** with some pattern, it is removed from the mailbox and its related actions are evaluated.
  - **`receive`** returns the value of the last expression evaluated in the actions.
  - If a **message does not match** with any pattern, it remains in the mailbox for further processing and processing continues with the next message in its mailbox.

Message reception

- The process that evaluates a **`receive`** is **suspended** until a message is matched.
- Messages arriving at a process **cannot block** other messages for that process.
- The **mailbox can be filled** with messages that do not match the patterns.
- Is the **responsibility of the programmer** to ensure that the mailbox does not fill up.

Specific process messages

- When you want to receive messages from a specific process, the sender must send its own **pid** in the message.

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

- The function **self/0** returns its pid to the calling process.
- This message can be received by

```
receive
    {Pid, Msg} ->
    ...
end
```

- Allowing to receive messages only from this process.

OTHER EXAMPLE

```erlang
-module(pingpong).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
        Pong_PID ! finished,
        io:format("Ping finished~n", []);
ping(N, Pong_PID) ->
        Pong_PID ! {ping, self()},
        receive
                pong -> io:format("Ping receives pong~n", [])
        end,
        ping(N - 1, Pong_PID).

pong() ->
        receive
                finished -> io:format("Pong finished~n", []);
                {ping, Ping_PID} ->
                        io:format("Pong recives ping~n", []),
                        Ping_PID ! pong,
                        pong()
        end.

start() ->
        Pong_PID = spawn(pingpong, pong, []),
        spawn(pingpong, ping, [3, Pong_PID]).
```

Another example

```
2> pingpong:start().
Pong receives ping
Ping receives pong
<0.36.0>
Pong receives ping
Ping receives pong
Pong receives ping
Ping receives pong
Ping finished
Pong finished
```

Timeouts

- The receive primitive can include **timeouts** so as not to block the process forever if it doesn't receive a message.

- **Syntax:**

```
receive
    Message1 [when Guard1] ->
          Actions1 ;
    Message2 [when Guard2] ->
          Actions2 ;
    ...
after
    WaitExpr ->
          WaitActions
end
```

Timeouts

- **`WaitExpr`** evaluates to an integer interpreted as a time in **milliseconds.**
- The **`WaitActions`** are evaluated if a message is not matched before the timeout expires.
- **Example**: suspend a process T milliseconds.

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

Another Example: detect double clicks

```
get_event() ->
      receive
            {mouse, click} ->
                  receive
                        {mouse, click} ->
                              double_click
                  after
                        double_click_interval() ->
                              single_click
                  end
            ...
      end.
```

Special waiting times

- There are 2 special waiting times:
  - **infinity**: Specifies a wait that will never occur.
    - Useful if the waiting time is calculated in real time (outside the **`receive`**).
  - **0**: specifies that the wait ends immediately.
    - But first, the system treats all messages currently in the mailbox.

Example

- To delete all messages from a process inbox

```
clear_mailbox() ->
     receive
          _ -> clear_mailbox()
     after 0 ->
          true
     end.
```

- Without the wait 0 would block until there was some message to delete.

Another example of wait 0

- Implement a form of reception with priorities.

```
priority_reception() ->
      receive
            {urgent, X} ->
                  {urgent, X}
      after 0 ->
            receive
                  Anyone ->
                        Anyone
            end
      end.
```

Process registration

- The PID of a process is required to send a message to it.
  - This is **very secure**, but **inconvenient** because the process must send its PID to all the other processes that want to communicate with it.
- Erlang has a **method to publish PIDs** so that any process in the system can send messages to them.
- The method is known as **process registration.**

Process registration

- **Predefined Primitives:**
  - `register(Alias, Pid)` – records the process **`Pid`** with the name **`Alias`** (an atom).
  - `unregister(Alias)` – removes any record with the name **`Alias`**.
  - `whereis(Alias)-> Pid | undefined` – determines if the name **`Alias`** is registered.
  - `registered()` – returns a list with all the processes registered in the system.

Example of process registration

```
-module(clock).
-export([start/2, stop/0]).
start(Time, Function) ->
    register(clock, spawn(fun() ->
        tictac(Time, Function) end)).
stop() -> clock ! stop.
tictac(Time, Function) ->
        receive
                stop -> void
        after Time ->
                Function(),
                tictac(Time, Function)
        end.
```

Process Record Example

- Make the clock tick and display the timestamp every 2 seconds:

```
3> clock:tictac(2000, fun() ->
io:format("TICTAC ~p~n",
[erlang:system_time(second)]) end).
true
TICTAC 1619806360
TICTAC 1619806362
TICTAC 1619806364
TICTAC 1619806366
4>clock:stop().
stop
```