SPP (Synchro et Prog Parallèle)

Unit 8: Immutability & Actors

François Taïani

Questioning Locks

- Why do we need locks on data?
  - ➔ because concurrent accesses can lead to wrong outcome
- But not all concurrent accesses problematic
  - ➔ concurrent writes -> potential corruption
  - ➔ concurrent reads + 1 write -> potential inconsistent obs
  - ➔ concurrent reads (no writes) -> OK
- Hence the idea
  - ➔ avoid the writes!
  - ➔ the extreme way: functional programming
  - ➔ less extreme: immutability

Functional Programming

- Lambda calculus in a readable form
- Purely functional languages
  - ➔ mainly Haskell (www.haskell.org)
  - ➔ functions = functions in mathematical term
  - ➔ one input -> always same output
  - ➔ no side effect (no “hidden state” that is modified)
- Implications
  - ➔ variables and their content are immutable
  - ➔ once value has been assigned, cannot change
- Wide ranging influence: see HFS, GSF
  - ➔ Hadoop file system, Google File System

Other Functional Languages

- With side effects / mutable fields in special cases
  - ML
  - OCaml (originally derived from ML)

- Hybrid
  - mix ideas of functional, imperative, and OO languages
  - the largest class
  - examples: Scala, Ruby, Erlang, F#

Example

- Haskell Factorial
  - `factorial n = if n==0 then 1 else n * factorial (n-1)`
  - once n bound in a scope (invocation), no longer changes

Interest for Concurrency

- Functional languages encourage stateless code
  - ➔ no state, no risk of corruption
  - ➔ also good in distributed implementation
- Immutable data
  - ➔ once produced, cannot change
  - ➔ so no risk of observing it in non-consistent state
  - ➔ no risk of having it corrupted
- Can also be applied to Java
  - ➔ **final** keyword: means field cannot change

Immutability in Java

- (Taken from Oracle Tutorial on Immutability)

```
final public class ImmutableRGB {
    final private int red;
    final private int green;
    final private int blue;

    public ImmutableRGB(int r, int g, int b) {
        this.red = r; this.g = green; this.b = blue;
    }
    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }
    public ImmutableRGB invert() {
      return new ImmutableRGB(255-red,255-green,255-blue);
    }
}
```

Notes on Example

- `final public class ImmutableRGB {`
  - ➔ means cannot be subclass
  - ➔ so a ref typed w/ `ImmutableRGB` can't point to subclass
  - ➔ dangerous: possibly change methods, add fields
- `public ImmutableRGB(int r, int g, int b)`
  - ➔ constructor: only place where state is set
  - ➔ rest of methods: only getters, no setters
- `final private int red;`
  - ➔ int basic type: direct value, not reference
  - ➔ with objects: must make sure they too are immutable

Does it solve all problems?

- No
  - ➔ only `ImmutableRGB` is thread safe
  - ➔ but can be used in ways that are not

```
ImmutableRGB a,b,c;
a = new ImmutableRGB(0,0,0);
b = new ImmutableRGB(255,0,0);
c = a;
a = b;
b = c;
```

Actors

- Inspired from CSP notion of Processes
  - ➔ Communicating sequential processes (1978)
  - ➔ proposed by Hoare (the same as Hoare's semantic)
  - ➔ CSP: no shared data / message passing
- Actors (Scala, Erlang)
  - ➔ system = a finite set of actors
  - ➔ actors interact through messages only (no shared mem)
  - ➔ messages are asynchronous (queued)
  - ➔ messages treated by message handlers
  - ➔ one handler at a time at most active in each actor

Actors in Erlang

- Called “Processes” (like in CSP)
- Directly built into the language
- Used for concurrency and distribution

Example

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

- Entry method for pong processes
  - ➔ prepare itself to receive either “finished” or “ping”
  - ➔ if ping, sends a pong message to Ping_PID

Example (cont.)

```
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 received pong~n", [])
    end,
    ping(N - 1, Pong_PID).
```

- methods for ping process
  - ➔ functional: no state, no loop, replaced by recursion
  - ➔ sends a ping message, waits for pong reply

Example (finish)

```
start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

- "main" program: entry point
  - ➔ starts 2 processes ("tut15" = name of package)
  - ➔ pong and ping = entry points of processes
  - ➔ [..] = parameters passed to entry points
- Similar to threads! but
  - ➔ no shared data
  - ➔ only interaction: asynchronous messages

Summary

- Alternative approaches to parallelism
  - ➔ immutability
  - ➔ actors
- Immutability
  - ➔ use immutable data structures
  - ➔ main philosophy of functional languages
  - ➔ wide ranging influence: HFS, GFS (Hadoop, Google)
- Actors
  - ➔ no shared data
  - ➔ only interaction: message passing
- Strong link to functional languages!