The FLK project

My first idea was to write a L4 kernel using the EIFFEL langage.

But ...

➀ While reading references on the SECURITY topic, it became evident that handling security by the software was an interesting challenge.

➁ It is difficult to find examples of secure kernel using the FLAT memory model (for processors without MMU – MMUs drain current –). That shows that there is an interesting challenge.

③ The thread/process model does not fit the SCOOP model because of the costs :

- in MEMORY (for stacks)
- in TIME (for context switches)

S imple
C oncurrent
O bject
O riented
P rogramming

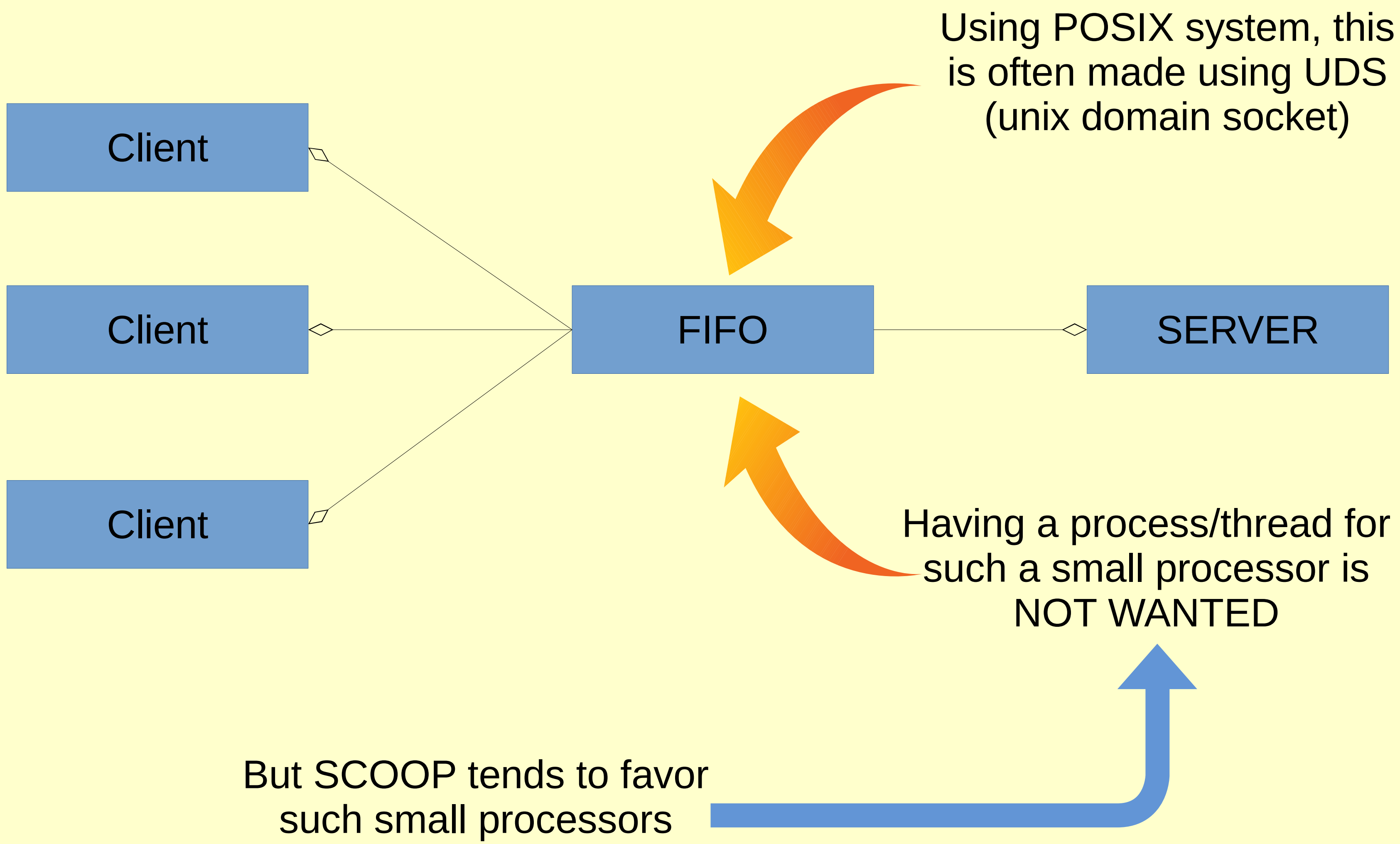
An overview...

Few terminology

- A **processor** is a set of objects within a same memory manager and with the guaranty that methods of the objects are executed “as single threaded”
- Objects of a processor can invoke object of other processors, such “remote” objects are called **separate** objects, they belong in a **separate** processor

```
-- DO YOU KNOW EIFFEL LANGUAGE ?
-- this is a simple FIFO description

deferred class FIFO[X] feature
  empty: BOOLEAN deferred end
  full: BOOLEAN deferred end
  put(x: X) require not full deferred ensure not empty end
  item: X require not empty deferred end
  drop require not empty deferred ensure not full end
invariant
  at_least_one_element: not (empty and full)
end
```

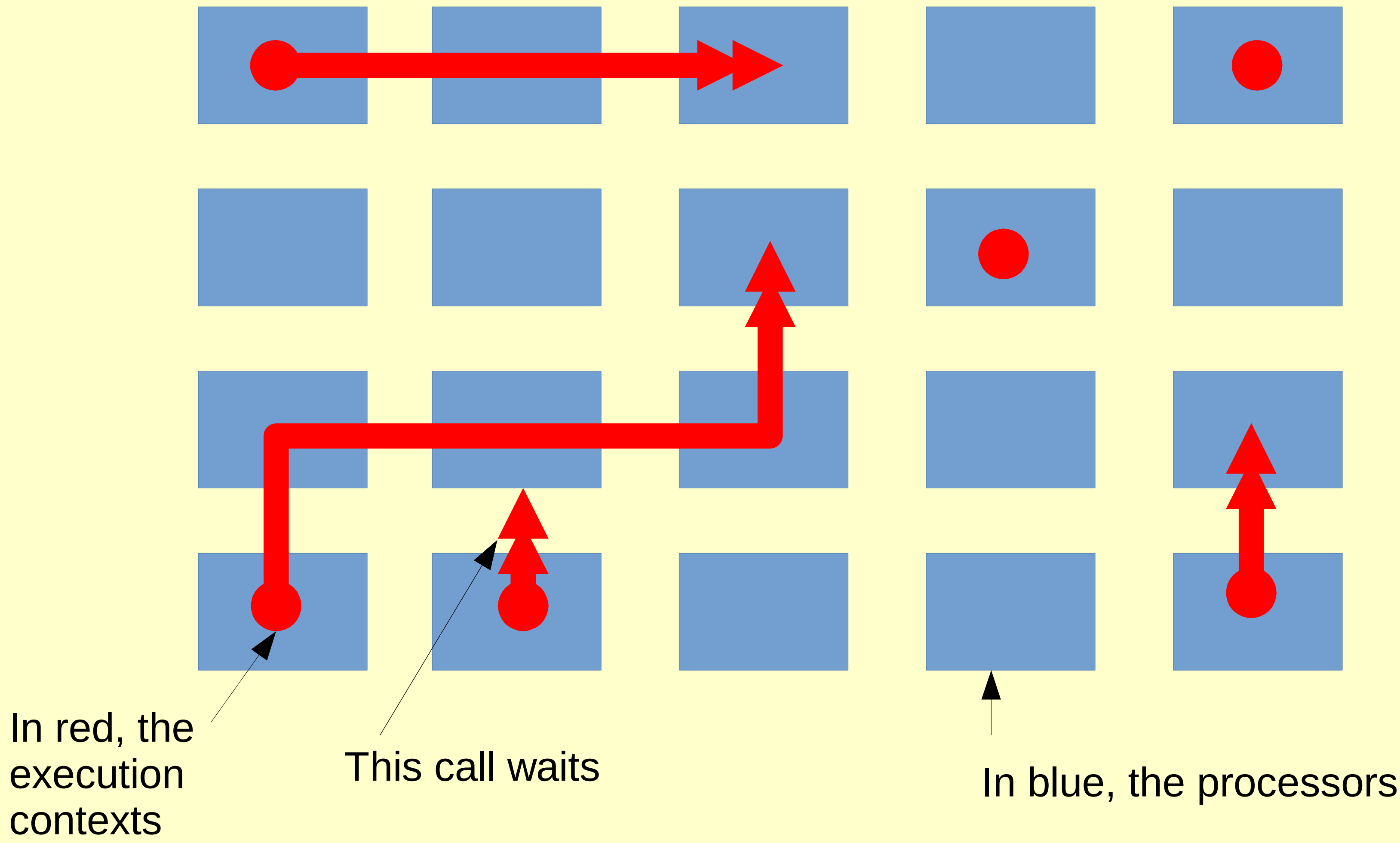
```
-- this is an abstract of a simple client
class CLIENT
  ...
  order_task(q: separate FIFO[separate TASK]; t: separate TASK)
    require not q.full do q.put(t) end
  ...
end

-- this is an abstract of a simple server
class SERVER
  ...
  next_task(q: separate FIFO[separate TASK]): separate TASK
    require not q.empty do Result := q.item ; q.drop end
  ...
end
```


Using POSIX system, this is often made using UDS (unix domain socket)
Client
Client
Client
FIFO
SERVER
Having a process/thread for such a small processor is NOT WANTED
But SCOOP tends to favor such small processors

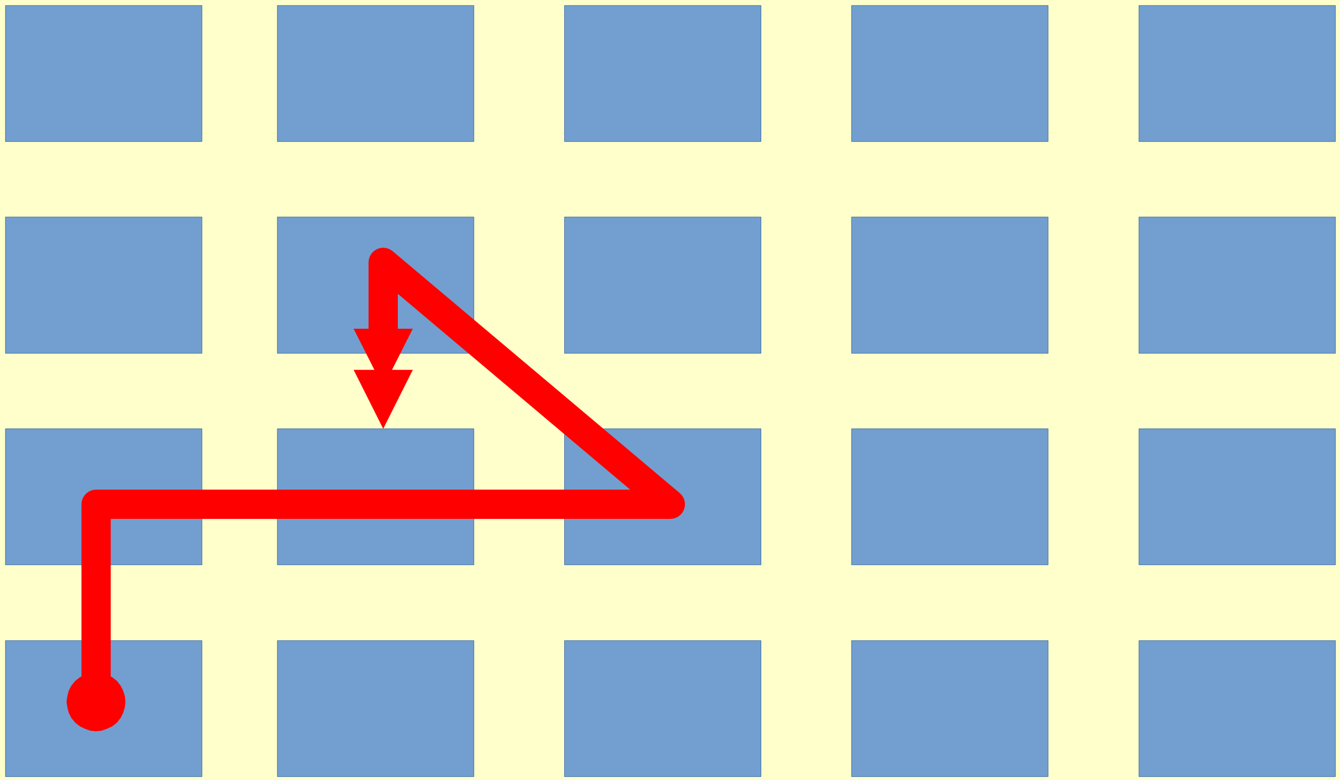
FLK principle

- Processors are nor processes neither threads
- Processors gain their single execution context on need and release it when not more need
- When invoking a separate object:
  - Either, the current execution context is lent to the called processor in case of query (having a result)
  - Or, a new execution context is created in case of command (not having result)
- Optimizations can modify the previous rule

Summary


In red, the execution contexts
This call waits
In blue, the processors

A failure case



Acquiring many processors

```
exchg(a, b: separate C)
  local x: X
  do x := a.item ; a.put(b.item) ; b.put(x)
  ensure a.item = old b.item ; b.item = old a.item
end
```

- The acquiring of multiple processors is granted by the system and the compiler in a safe way
- An implementation compatible with distributed system (Rhee lock) can be used if needed

Safety

- EIFFEL language is safe:
  - No peek / poke
  - Array boundaries are checked
  - Calling void reference is not possible
  - Wrong casting of objects is not possible
  - The memory is managed (GC, no “free”)
- SCOOP is safe:
  - Separate objects are tracked and not alterable
- No IPC, no IDL: consistency by the compiler

Wait a minute.... FLK is written using EIFFEL thus how is it possible if the language doesn't allow peek/poke/casting?

EIFFEL provides selective exportation of features.

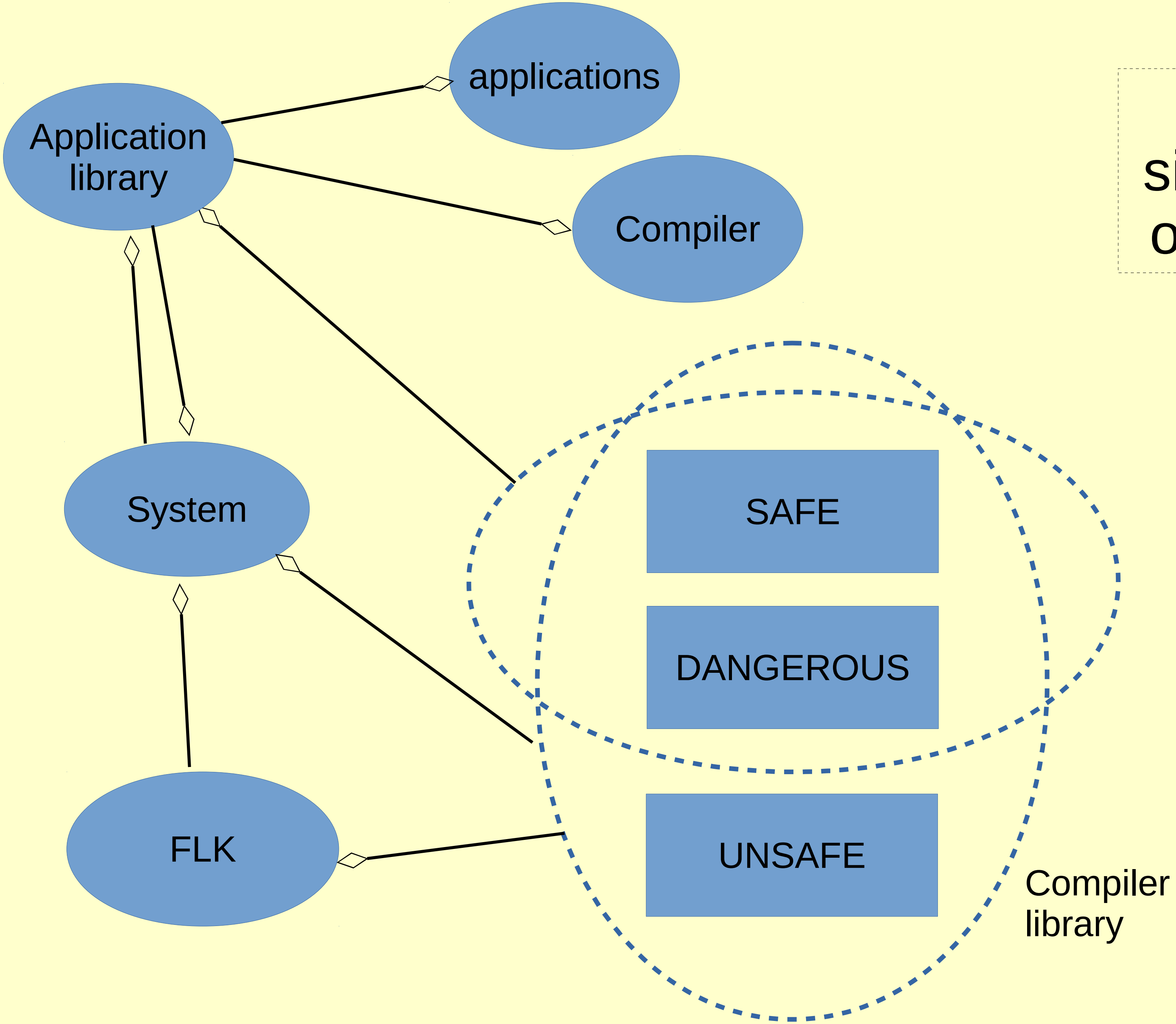
```
feature {UNSAFE_KERNEL}
  ... some unsafe features only for FLK ...

feature {UNSAFE_SYSTEM}
  ... some restricted features only for system components ...
```

This mechanism is enforced by the language and the compiler is improved to forbid inheritance of critical classes either outside of a cluster of classes or without integrator authorization

The compiler allow in fact peek/poke/casting but in a VERY RESTRICTED way

Security at API level

- Using the exportation feature of EIFFEL and the compiler FLC:
  - Features are selectively exported to specific client classes
  - Such authorized client classes can be:
    - not inheritable and not insertable outside a given context
    - Inheritable (or insertable) with integrator/user authorisation (when compiling or installing)
    - Inheritable or insertable as usual



Kind of simplified overview

MEMORY

- The memory is managed. That is in the language and it is done at the kernel level.
- Each processor has its memory manager
- Any pointer maps to a unique memory manager that maps to a unique processor: that is used to identify the processor of the separate objects
- Mechanic of the keyword **separate** and safety of the language allows to not protect memory using MMU

MMU?

- Using static analysis of code, the size need by the stack can be pre-computed for each processor entry method (recursivity...) thus MMU is not strictly needed for stacks
- But MMU can be useful for:
  - STACK
  - FILE MAPPING
  - LARGE DYNAMIC ARRAYS
  - FOREIGN LIBRARIES
- If used, MMU is global (not per processor)

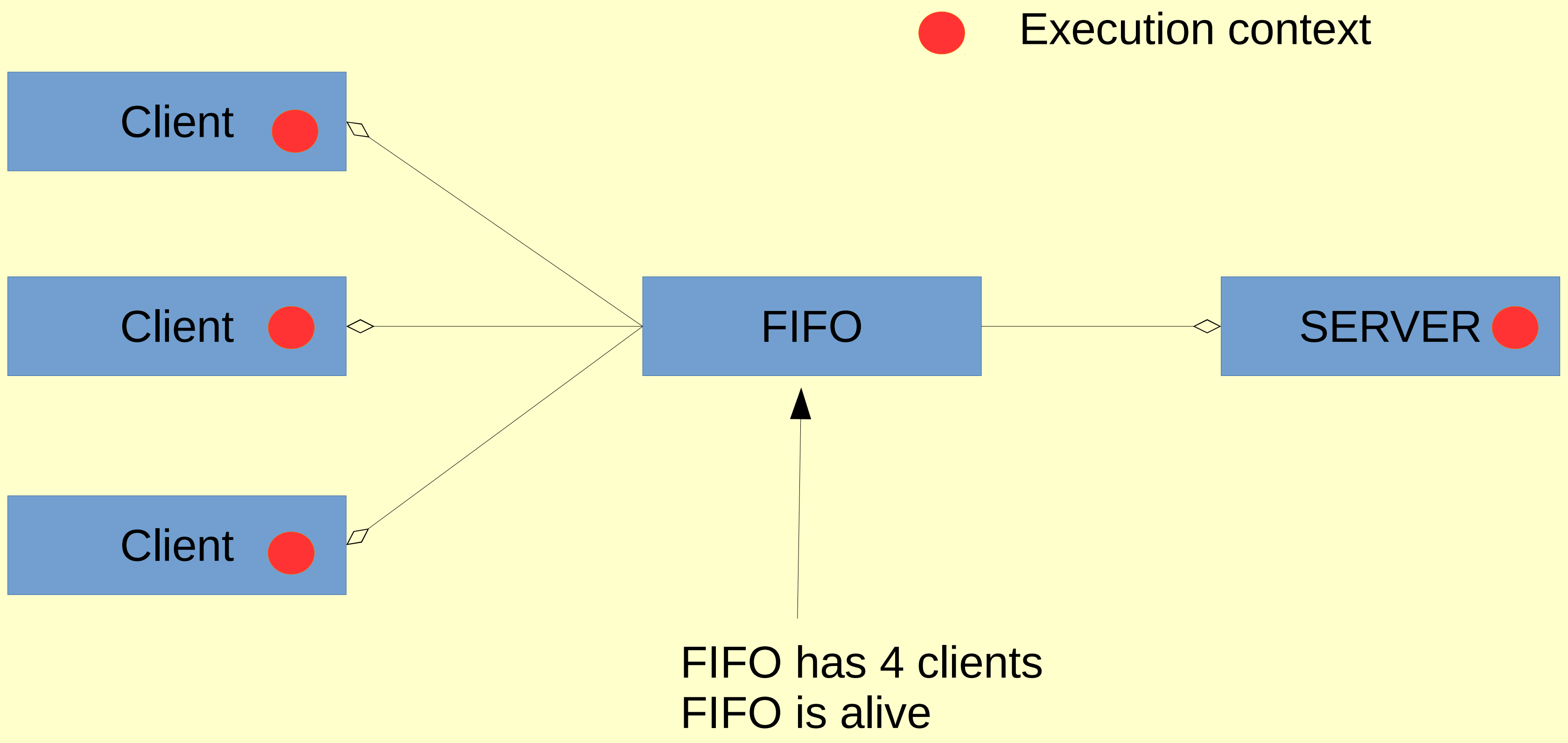
Context switch

When a method of a separate object is called, the execution context should:

- On a query:
  - Set the current processor to the one called, this activates its memory manager
  - Revert at end of the call
  - No task switch
- On a command:
  - Activate a new execution context for the called processor
  - Can be optimised
- No memory switch (MMU)

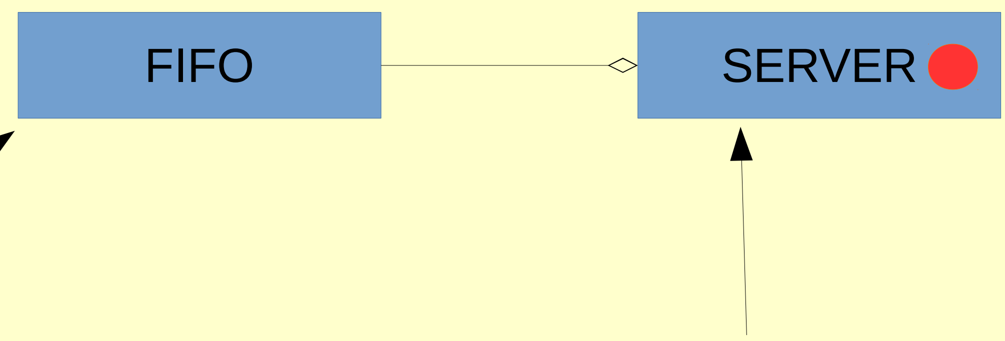
Life cycle

- Processors are created using the **create** keyword
- Processors die when
  - It is not in an execution context
  - It has no client (no other processor reference it)
- Processors are GARBAGE COLLECTED
- Starve conditions are detected and generate exceptions


Execution context
Client
Client
Client
FIFO
SERVER
FIFO has 4 clients
FIFO is alive

AFTER CLIENTS
DEATH

● Execution context



FIFO has 1 client
FIFO is alive

Alive because execution context
But waiting on

```
require not fifo.empty
```

Where FIFO has only one client
Thus, server is dead because
FIFO can't change

FLK SENDS AN EXCEPTION TO
THE SERVER TO SIGNAL THAT
ITS WAITING IS UNUSEFUL

Optimisation

- EIFFEL is making compilation of whole systems (no linking is needed)
- **Frozen** keyword allows more optimisation
- Some creation of execution contexts can be removed
- Unused components are removed from the binary set

Challenge 1: coupling of the memory management using compacting GC at kernel, system and application level

Challenge 2: improved security at API level without using capabilities (static)

Challenge 3: optimisation of activations, avoid creating a new context when possible (using frozen)

Challenge 4: allowing downsizing for tiny embedded environments

Challenge 5: create standard on semantic of **separate** and implements some of its tricky items

Challenge 6: avoid linker paradigm

Challenge 7: prove the safety, the security, the efficiency

BACK TO the ReaLiTY

How many drivers?

How many supported platforms?

How many code reusable?

How much money?

How many people?

planning

- Finish the compiler end of 2015, opening code ASAP
- RFCs process for FLK starting in spirng 2015
- First implementation of FLK on top of an other kernel in 2016
- Help wanted? YES YES YES
  - Coding, specifying, financing, research, students, how to integrate existing C drivers

QUESTIONS

sopox@flhq.org