Elixir |> Erlang

Concurrency Solution

September, 2021

TAM Le Duc

CPO |> iPlus
tam.le@iplus.vn

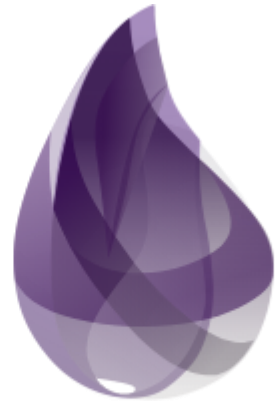
le_duc_tam

leductam

The land of Elixir |> Erlang

If you know Elixir, you love it most
You love Elixir, love Erlang too

SINCE 1986



elixir

SINCE 2011

RELIABILITY

CONCURRENCY

SCALABILITY

MAINTENANCE

ecto

ejabberd

verne MQ

FINCH

NERVES

Livebook

Phoenix Framework

MONGOOSEIM

RabbitMQ

BROADWAY
AN ELIXIR LIBRARY

HEX

Nebulex

poolboy

COWBOY

riak

Axon


aws


Bloomberg


Discord


cisco


Klarna.


VOCALINK
mastercard


bet365


WhatsApp


Pinterest


ERICSSON


Memo Bank


PEPSICO


Duffel


Goldman
Sachs


B/R
BLEACHER
REPORT

Concurrency vs Parallelism

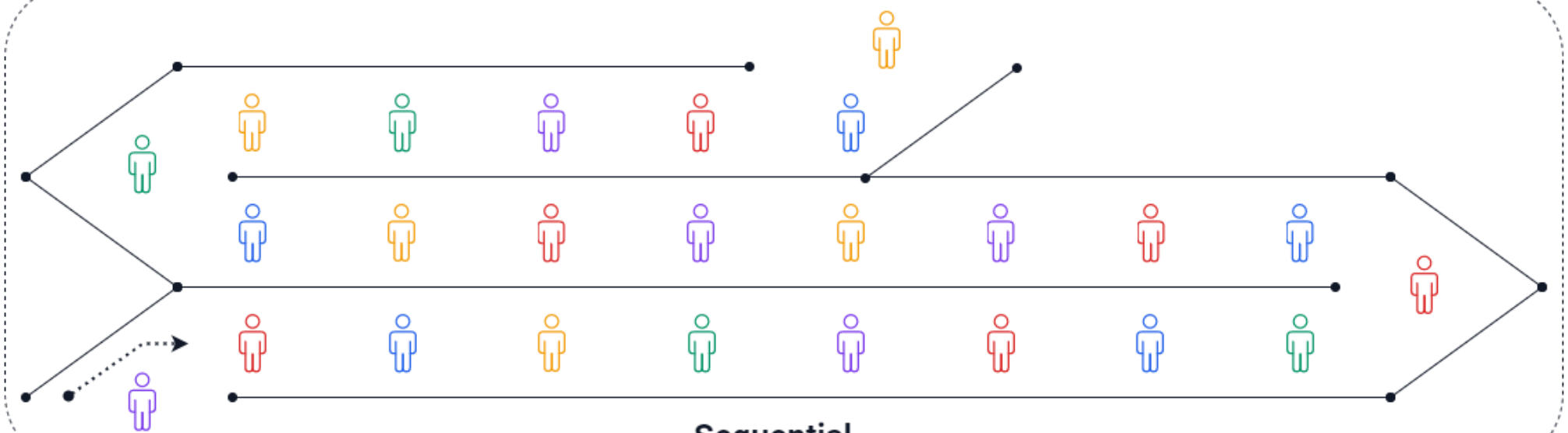
Get one-way ticket to galaxy of computer science

iPlus

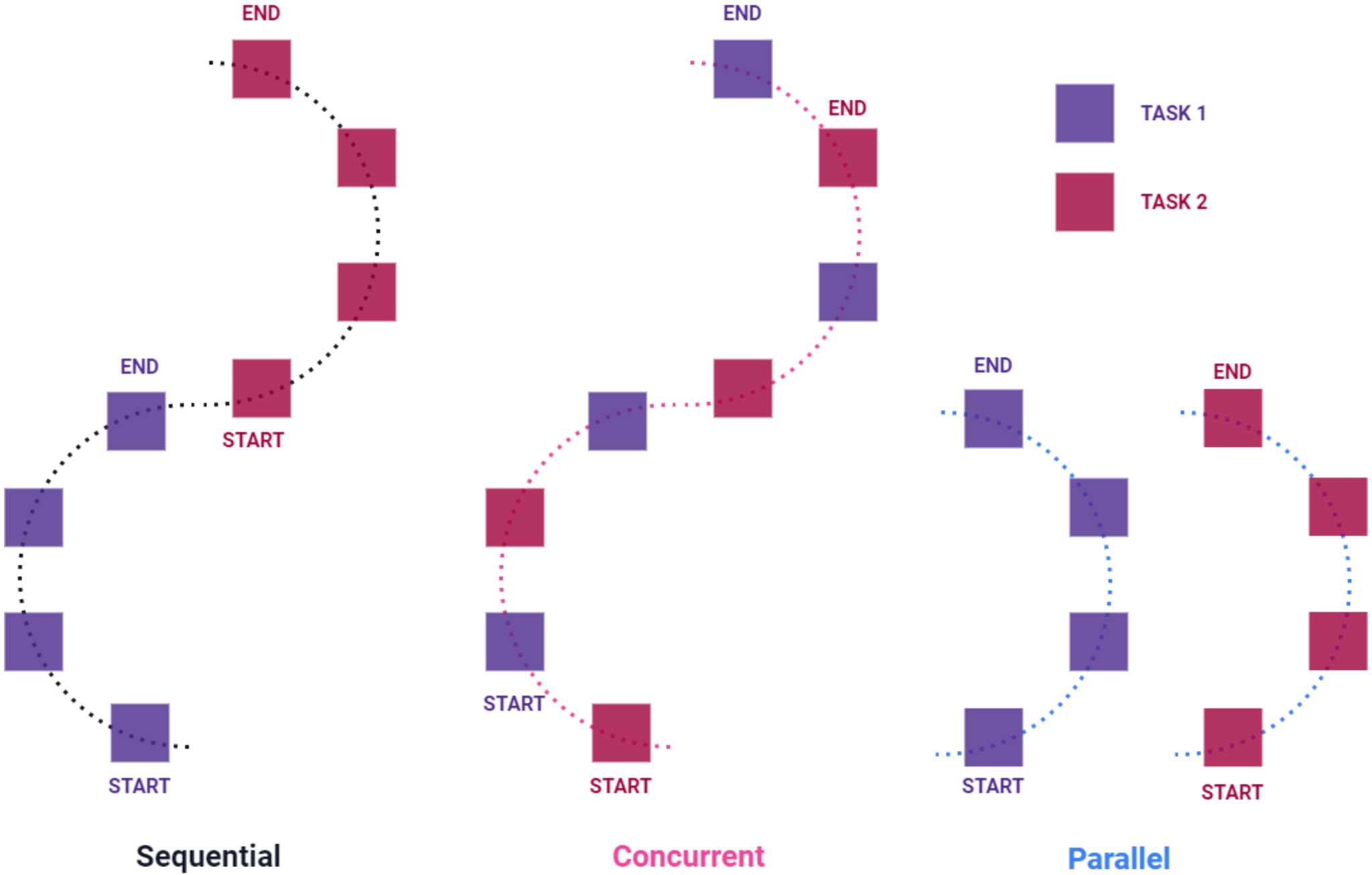
Concurrent

Space tourism check-in

Parallel

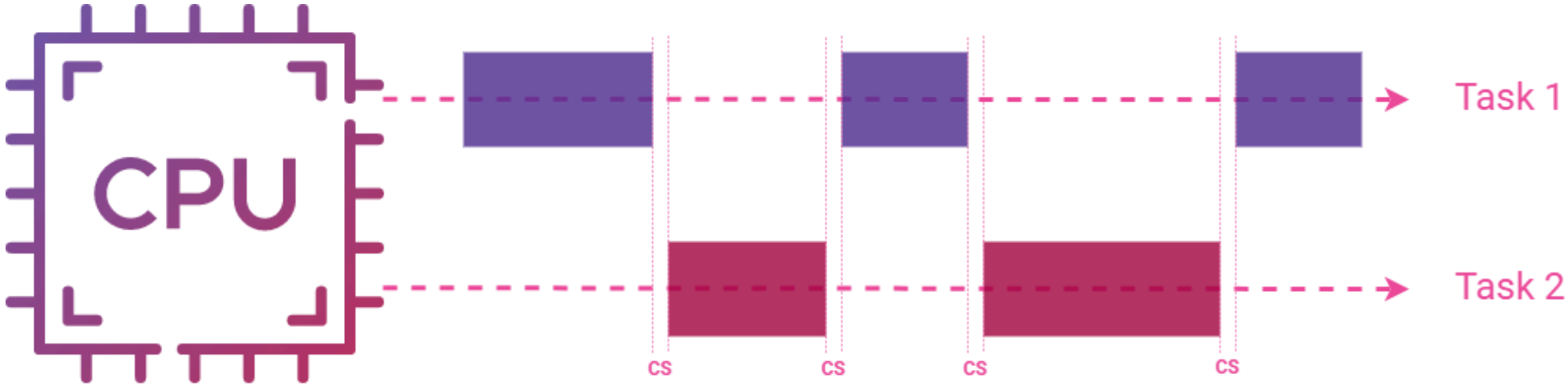


Sequential


END
END
START
START
END
END
START
START
TASK 1
TASK 2
END
START
END
START
Sequential
Concurrent
Parallel

Concurrency

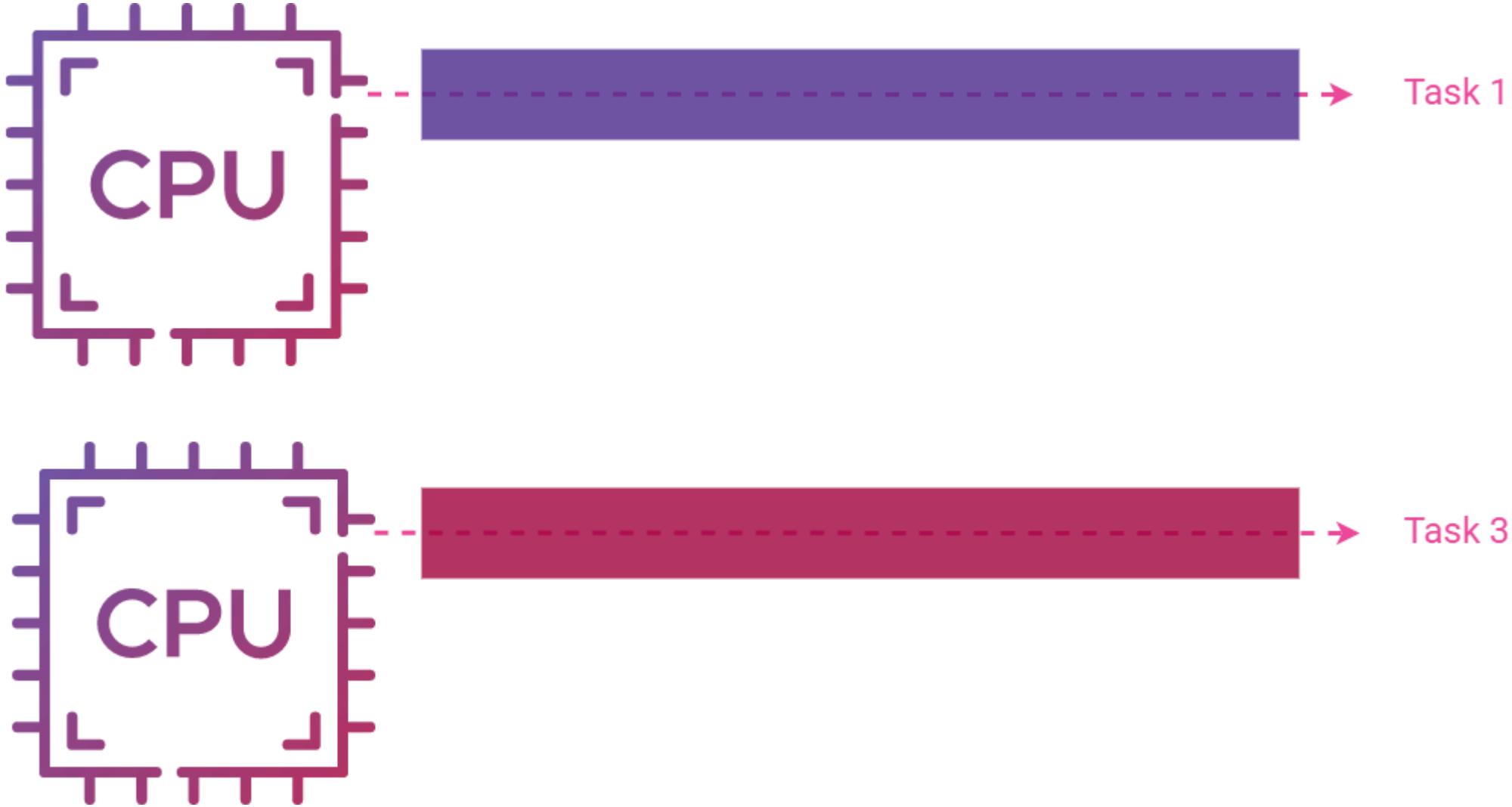
Concurrency is the composition of independently executing things. Concurrency is about dealing with lots of things at once, means that an application is concurrently making progress on more than one task at the same time (or at least seemingly at the same time).
To make progress on more than one task concurrently the CPU switches between the different tasks during execution, which means multiple tasks can start, run, and complete in overlapping time periods.
Concurrency is obtained by interleaving operation of tasks on CPU or through *context switching*.



CS - Context Switching

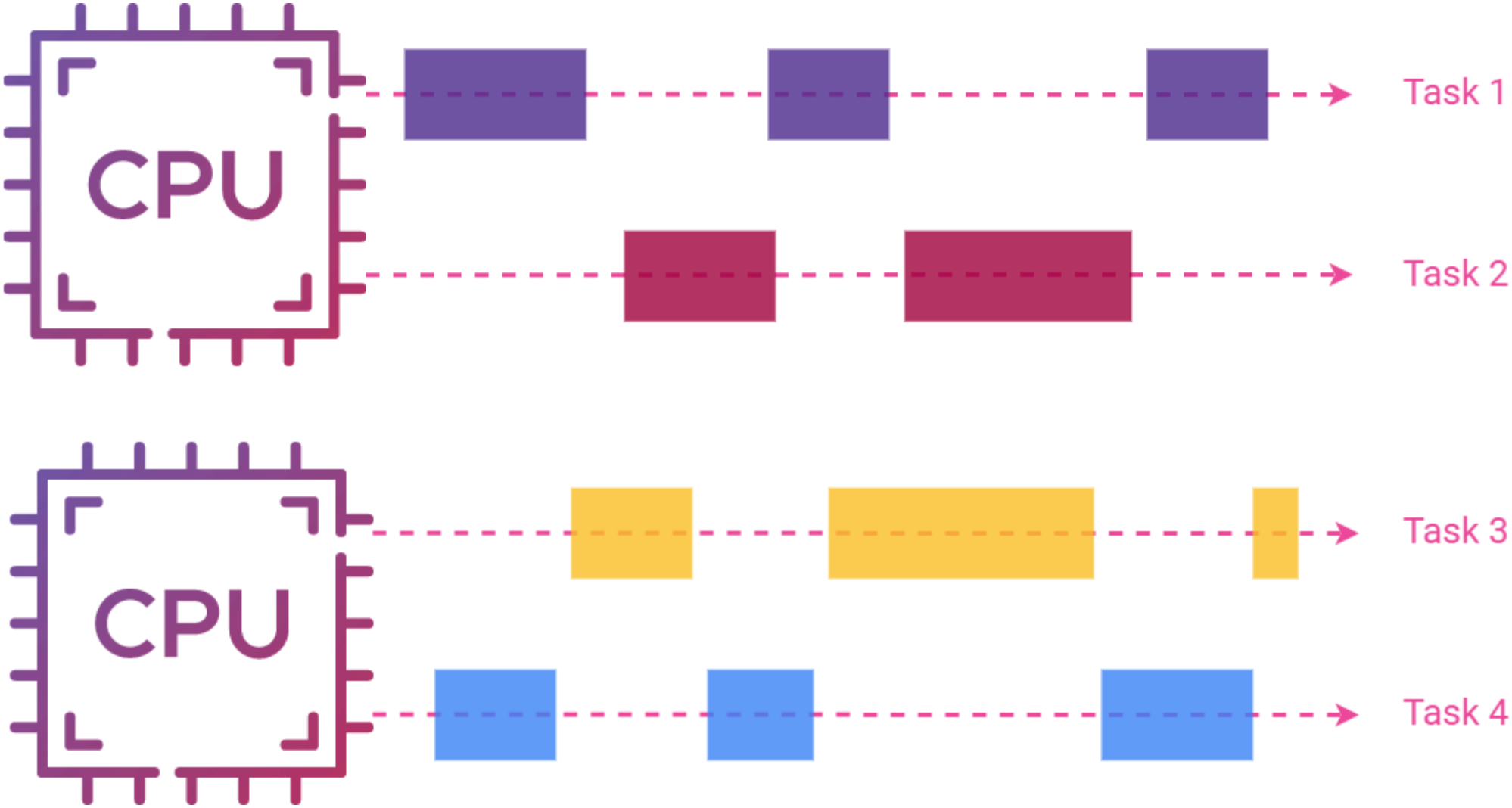
Parallel Execution

Parallel Execution is when a computer has more than one CPU, and makes progress on more than one task simultaneously.



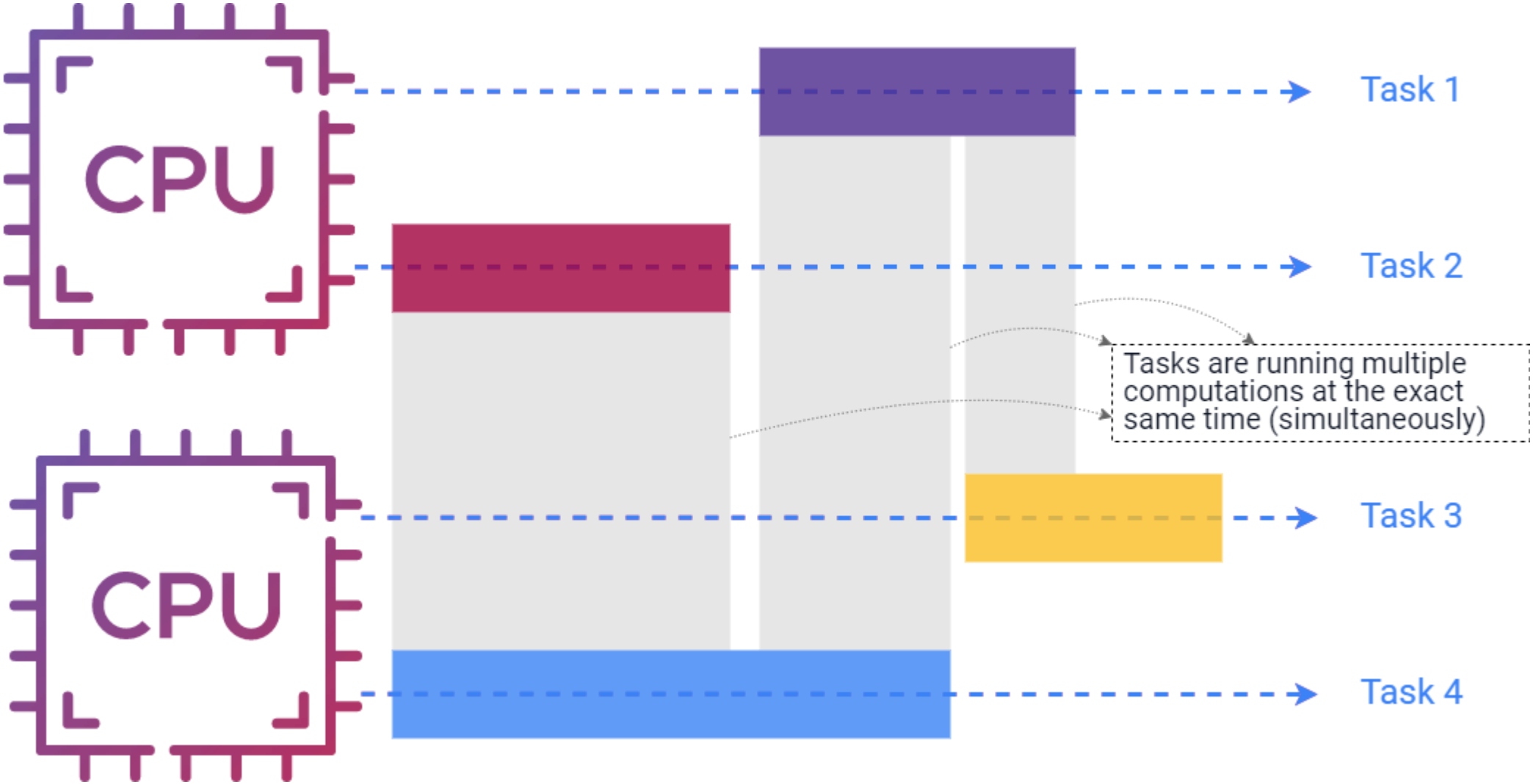
Parallel Concurrent Execution

Tasks are distributed among multiple CPUs, the taks executed on the same CPU are executed concurrently, whereas tasks executed on different CPUs are executed in parallel.



Parallelism

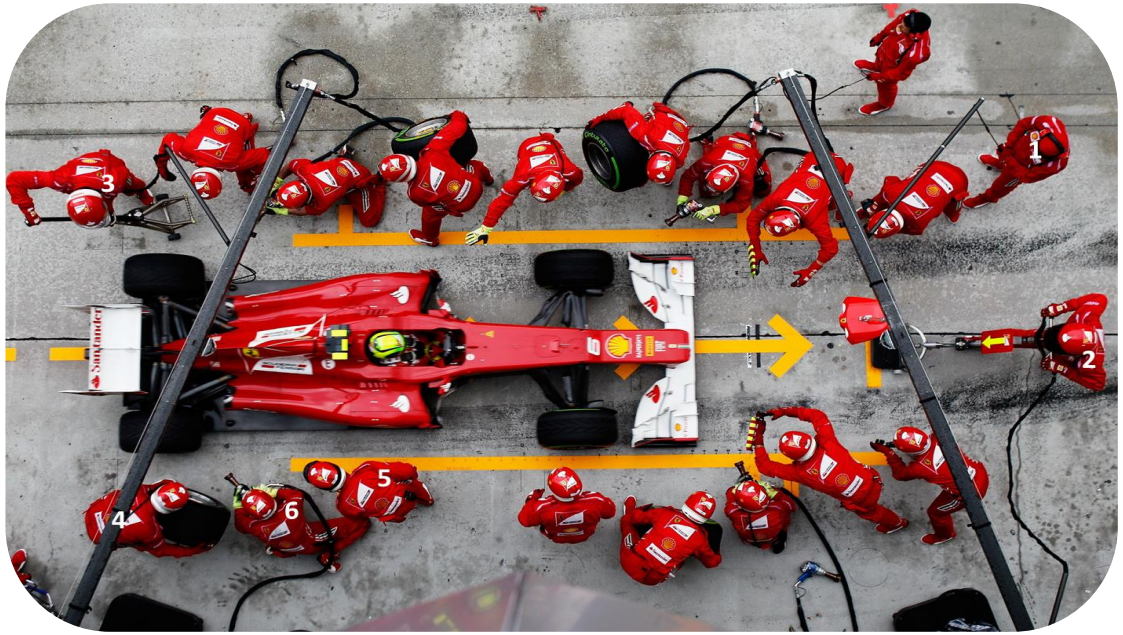
Parallelism is the simultaneous execution of multiple computations, mean that having the tasks (or subtasks) are running exactly at the same time on multiple CPUs.



Concurrency vs Parallelism

Comparison	Concurrency	Parallelism
Statement	Concurrency is the composition of independently executing things. Concurrency is about dealing with lots of things at once.	Parallelism is the simultaneous execution of multiple computations. Parallelism is about doing lots of things at once.
Archived through	Interleaving operation of tasks on the CPU or in other words by the context switching.	The tasks are simultaneously execute on multiple CPUs at exact same time.
Done by	Concurrency can be done by using a single processing unit.	Parallelism can't be done by using a single processing unit. It needs multiple CPUs.
Benefit	Increases the amount of work finished at a time.	Improves the throughput and computational speed of the system.
Example	Handle keyboard, mouse events. Clickable on the GUI button while the application saving files.	Counting character 't' in a big text file by splits simultaneous tasks on each chunk of the file.

Concurrency is an aspect of solution to deal with problem at design level that may (*but not necessarily*) execute in parallelism at run-time by taking the advantage of multiple cores.

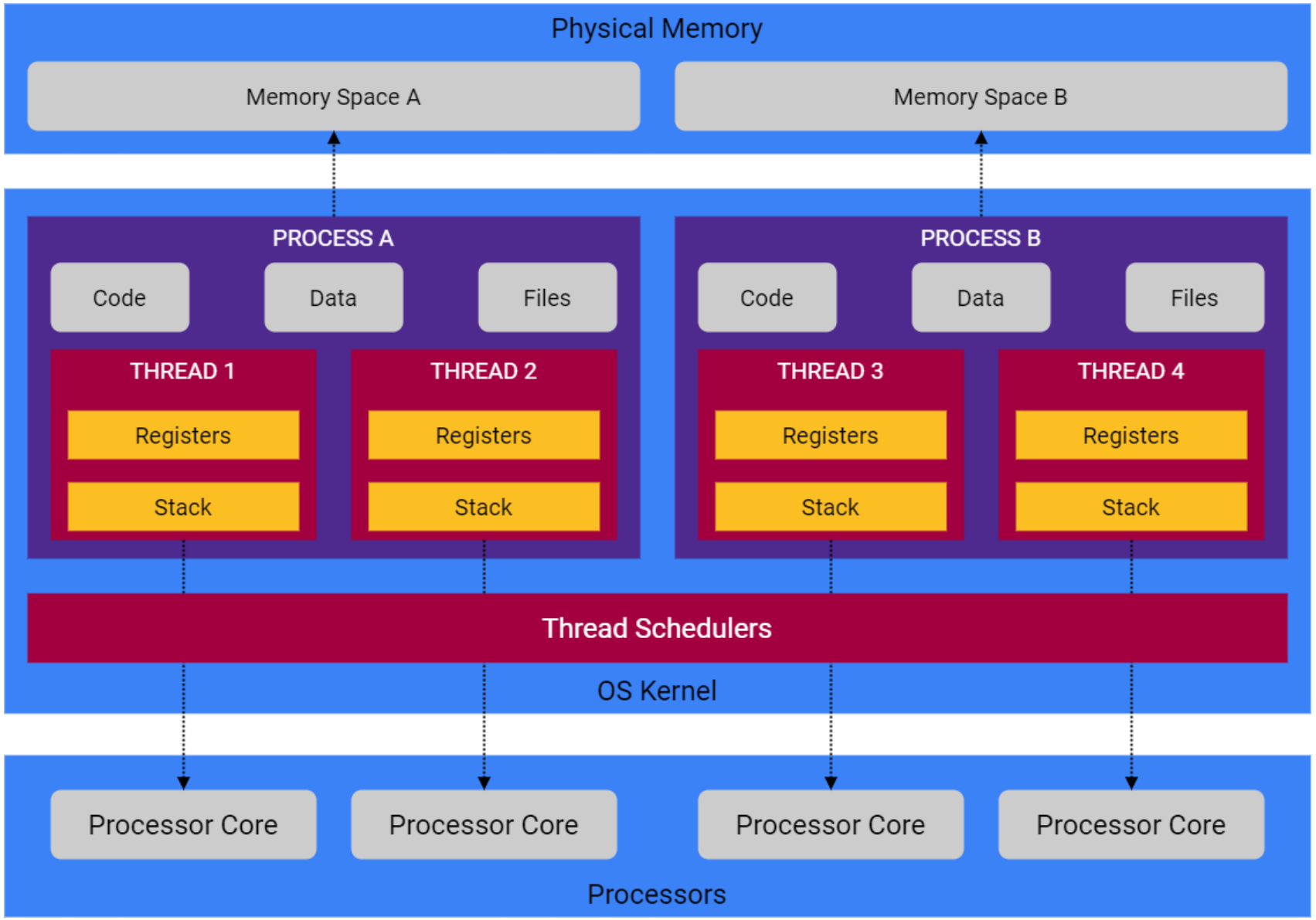



FIFA
VAR ROOM

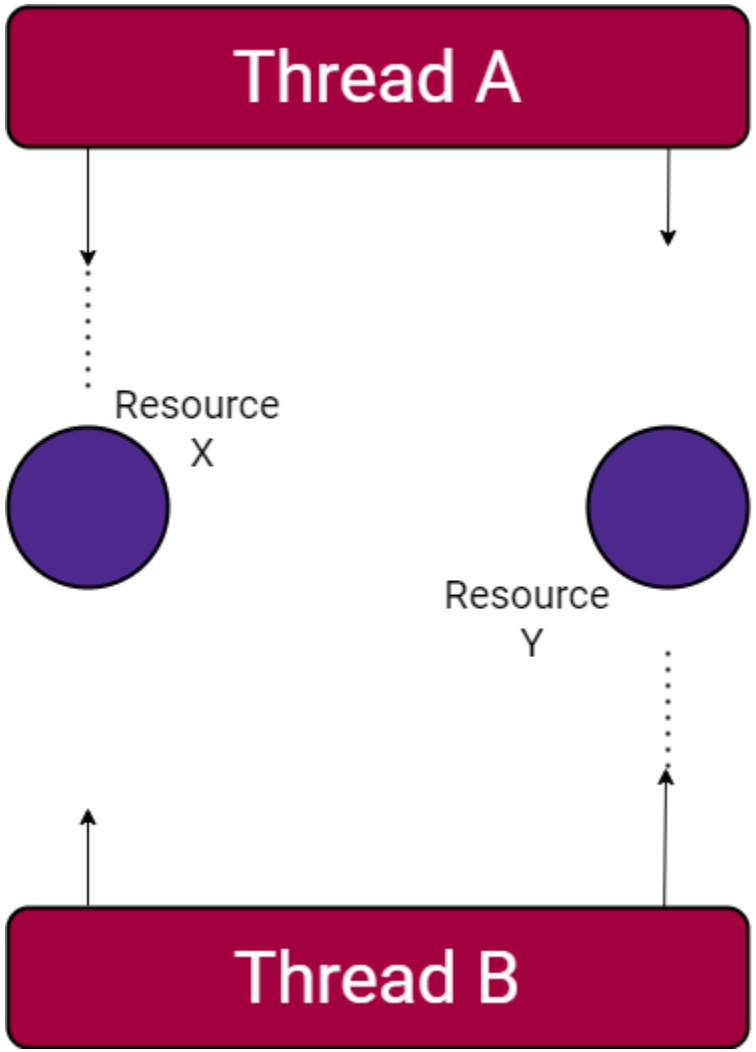
Concurrency Models

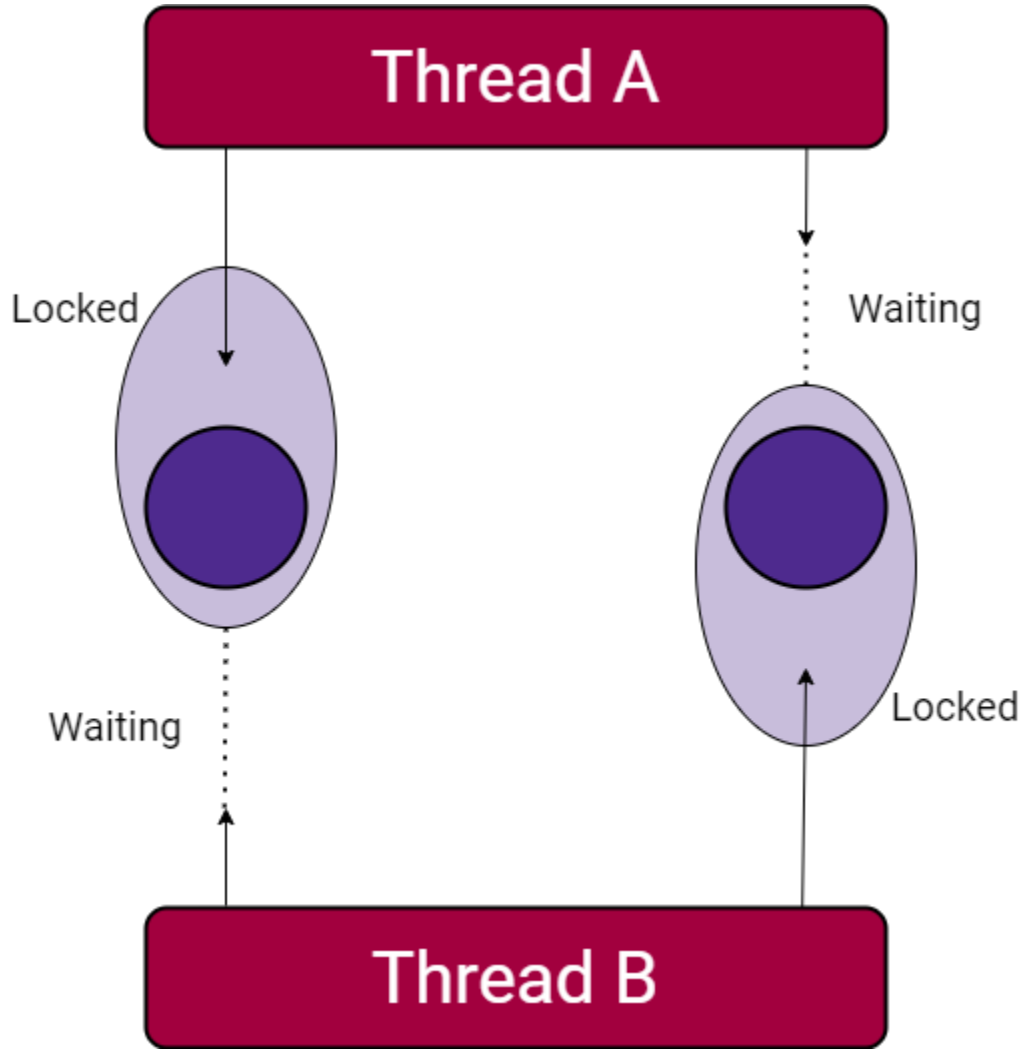
Threads and locks • Actors
Communicating Sequential Processes

Process & Thread



Threads and Locks

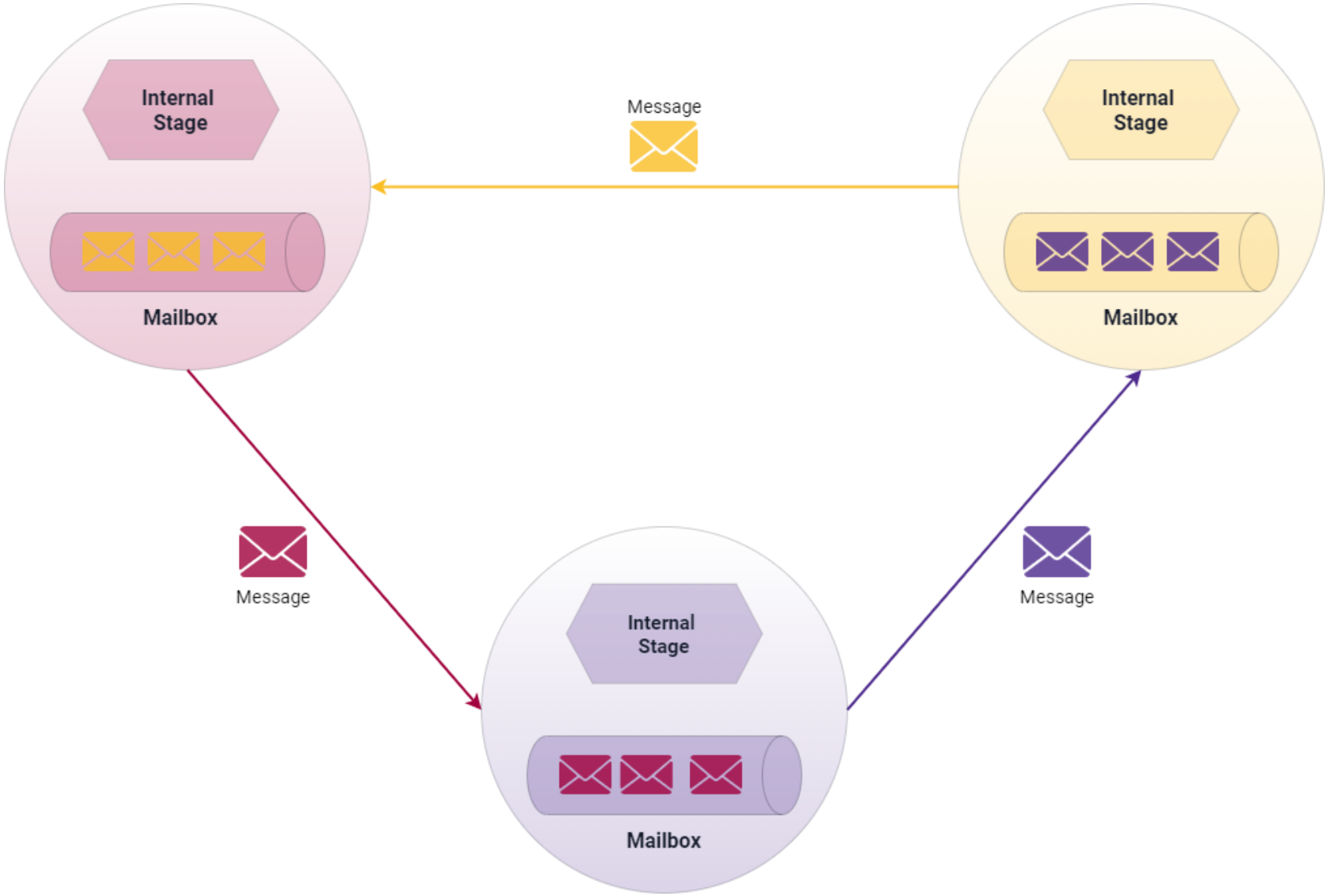




Threads and Locks

- Communicate by sharing memory
- Use locking mechanisms to prevent overwriting
- Dead locks, live locks, unpredictability
- Complexity to manage

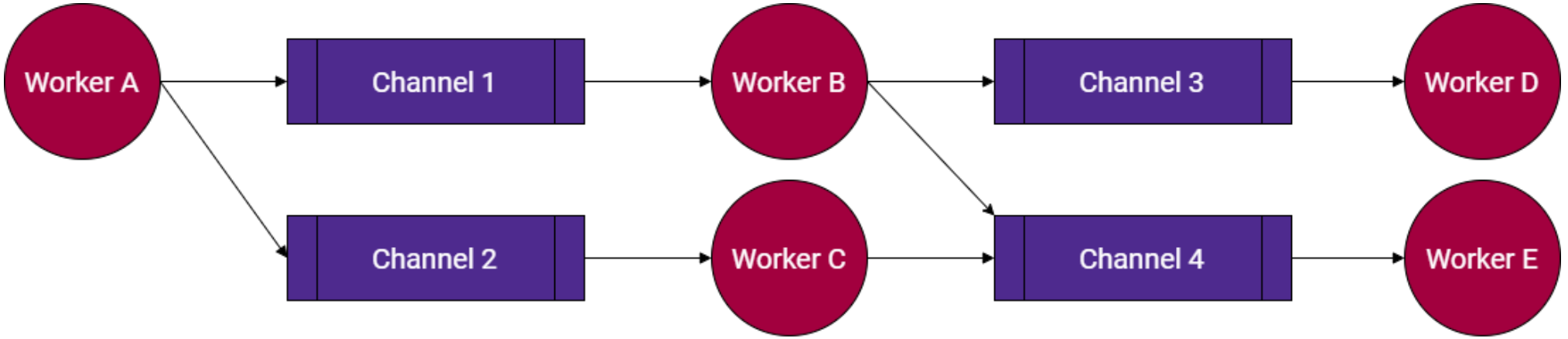
Actors Model



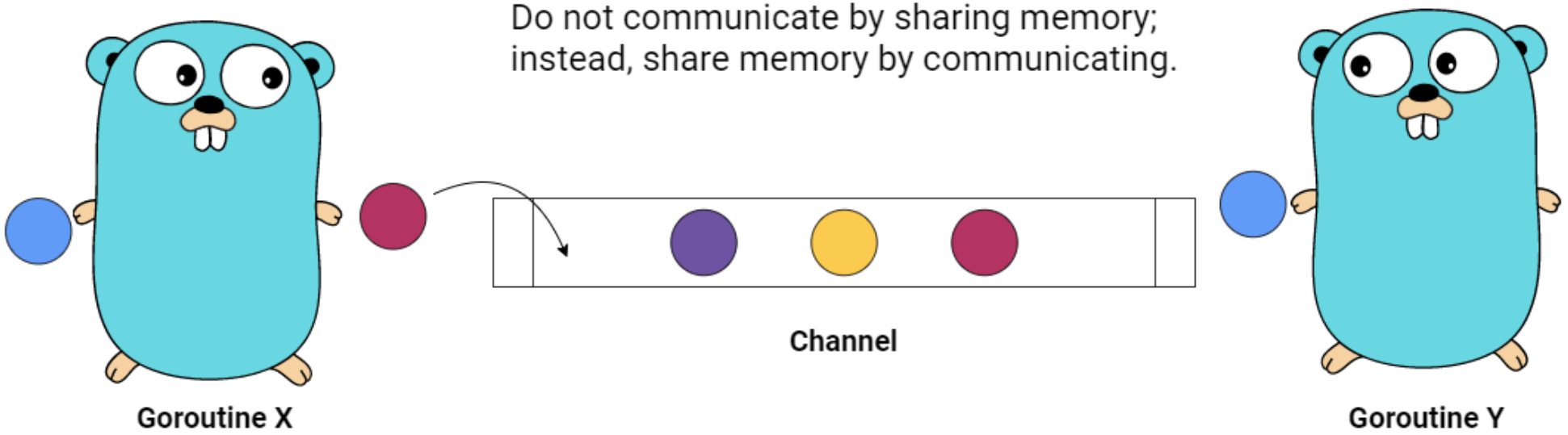
Actors Model

- Actors have the internal states and doesn’t share anything with others
- Actors communicate with each other by sending asynchronous messages. Those messages are stored in other actors' mailboxes until they're processed.
- When an actor receives a message, it can do one of these 3 things:
  - Handles the message and might update its state
  - Creates more actors
  - Sends messages to other actors

Communicating Sequential Processes

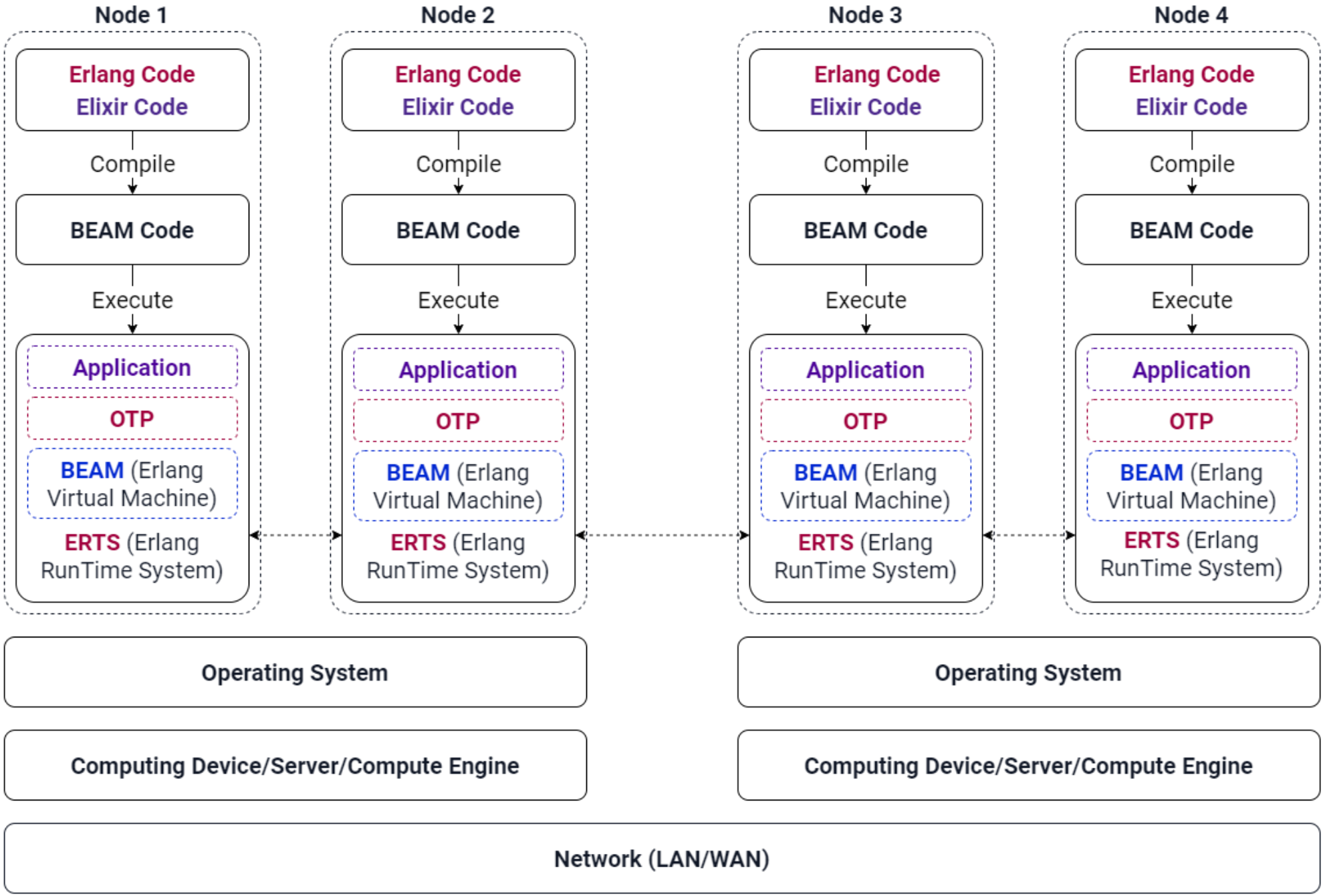


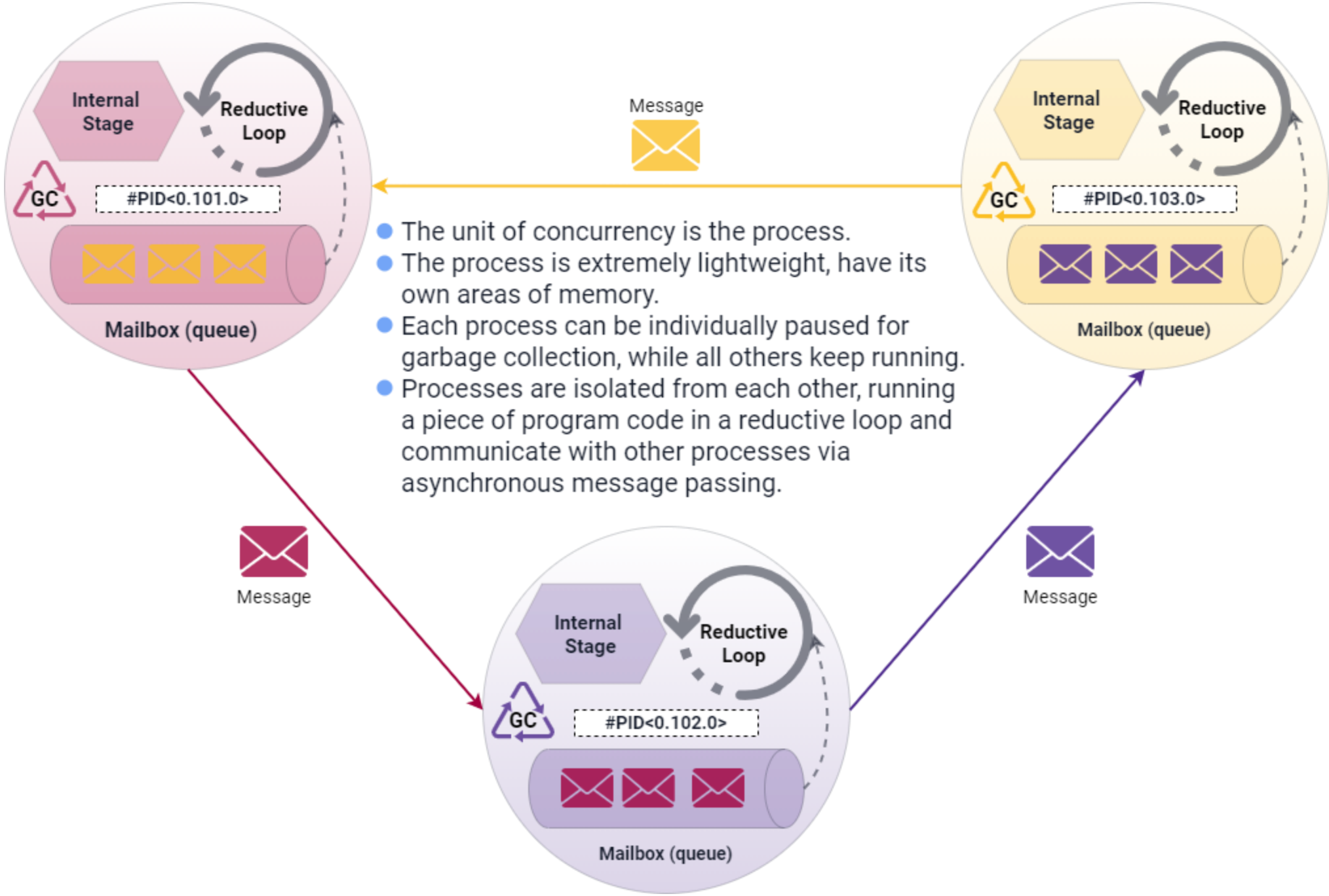
Do not communicate by sharing memory;
instead, share memory by communicating.

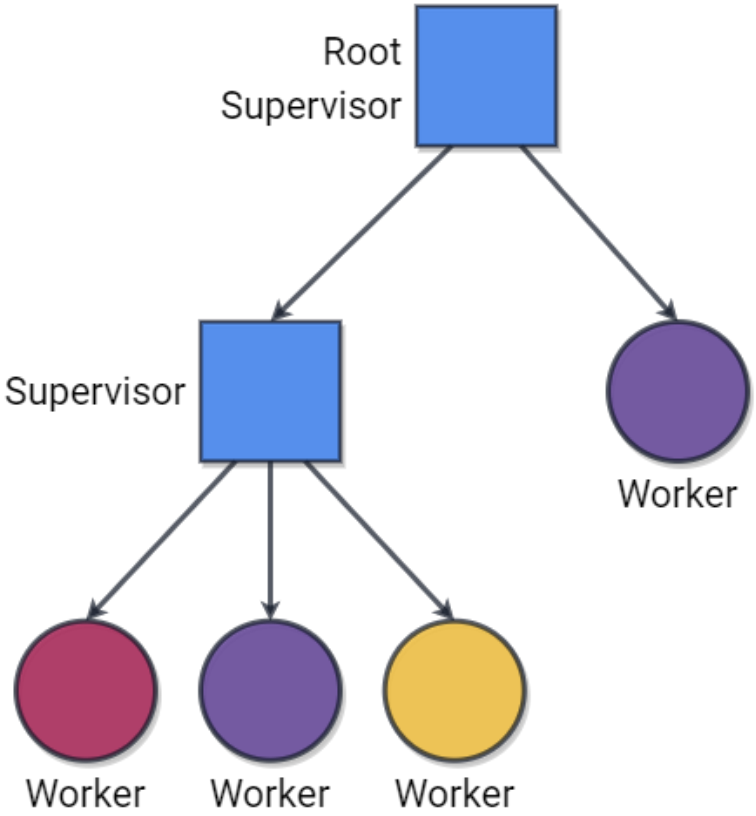


Concurrency in Elixir |> Erlang

Process model ● OTP ● Scheduler

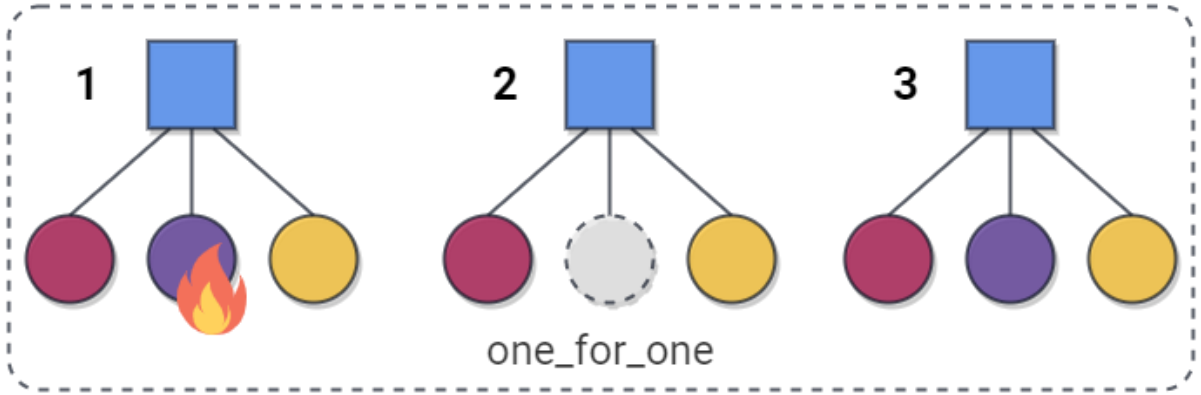

Node 1
Node 2
Node 3
Node 4
Erlang Code
Elixir Code
Compile
BEAM Code
Execute
Application
OTP
BEAM (Erlang Virtual Machine)
ERTS (Erlang RunTime System)
Operating System
Computing Device/Server/Compute Engine
Network (LAN/WAN)

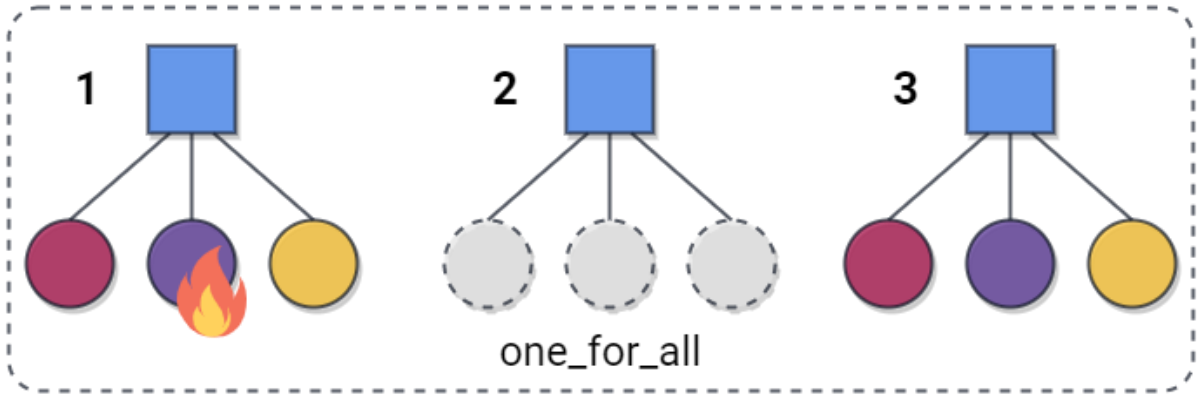

Internal Stage
Reductive Loop
GC
#PID<0.101.0>
Mailbox (queue)
Message
Internal Stage
Reductive Loop
GC
#PID<0.103.0>
Mailbox (queue)
The unit of concurrency is the process.
The process is extremely lightweight, have its own areas of memory.
Each process can be individually paused for garbage collection, while all others keep running.
Processes are isolated from each other, running a piece of program code in a reductive loop and communicate with other processes via asynchronous message passing.
Message
Message
Internal Stage
Reductive Loop
GC
#PID<0.102.0>
Mailbox (queue)

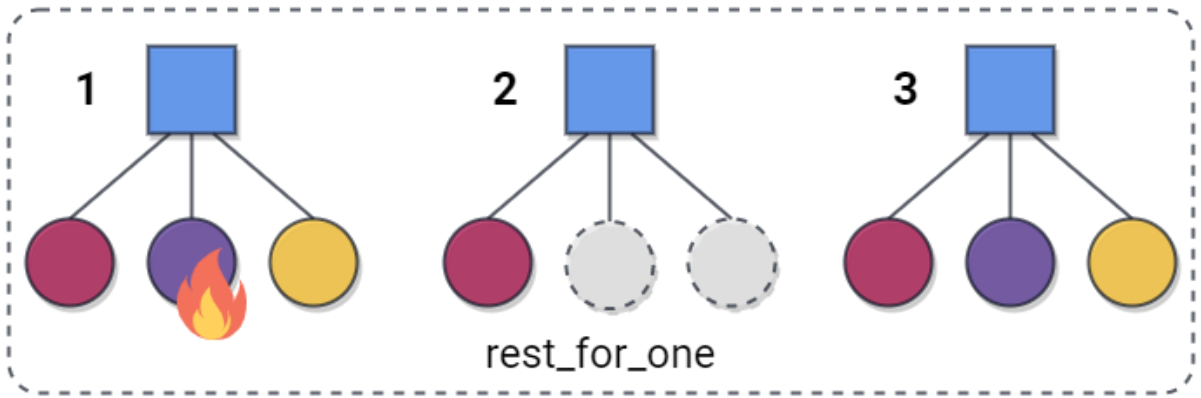


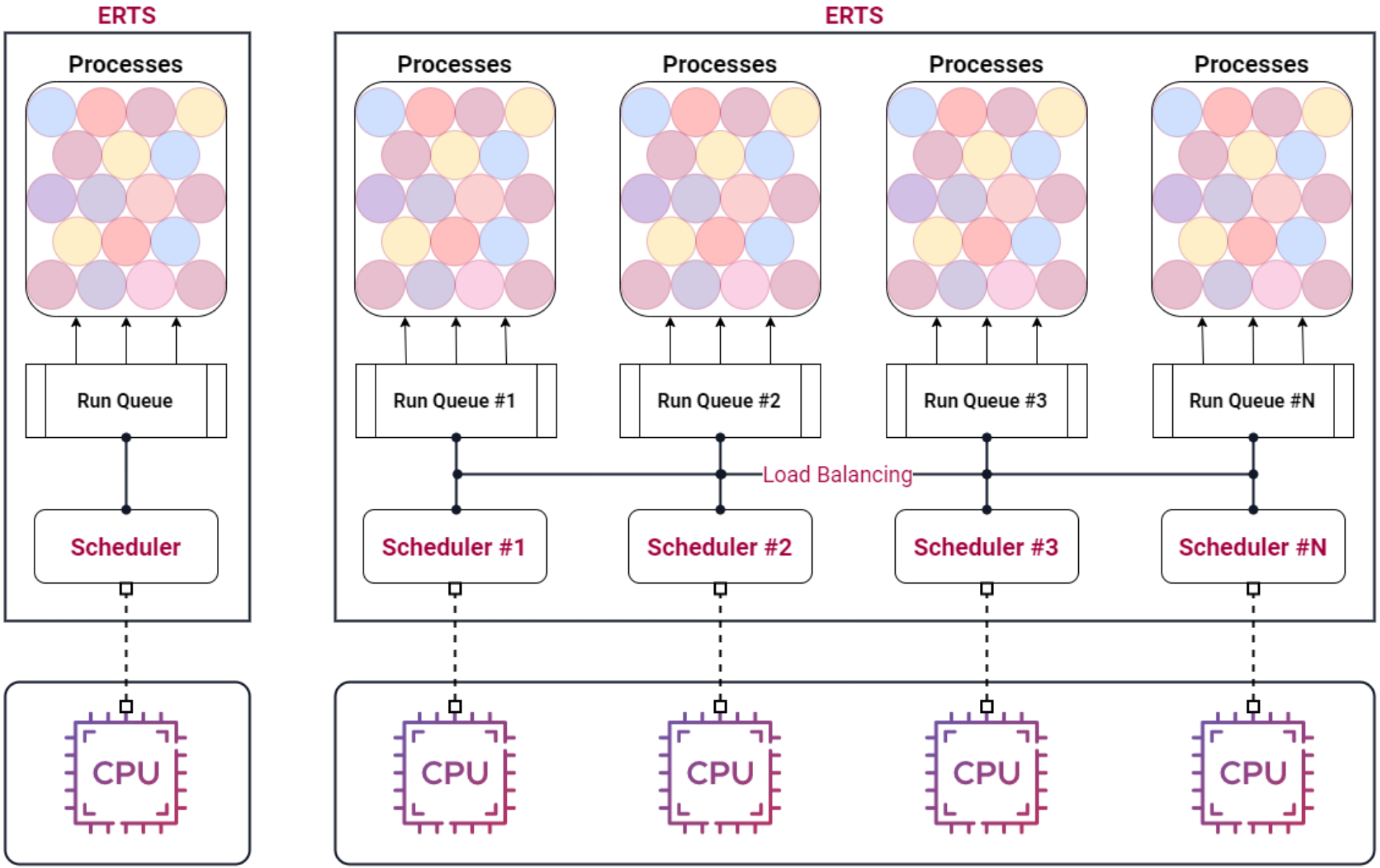
- Supervisors start child processes.
- Supervisos monitor their children.
- Supervisors can restart the children when. they terminate.

- OTP is a set of frameworks, principles, and patterns to structure, design, implement, and deploy the concurrent, fault-tolerant systems.









Processes on uniprocessor.

Processes on multiprocessor. The runtime system automatically load balancing the workload over schedulers bind to the available CPU resources.

Concurrency in Elixir |> Erlang

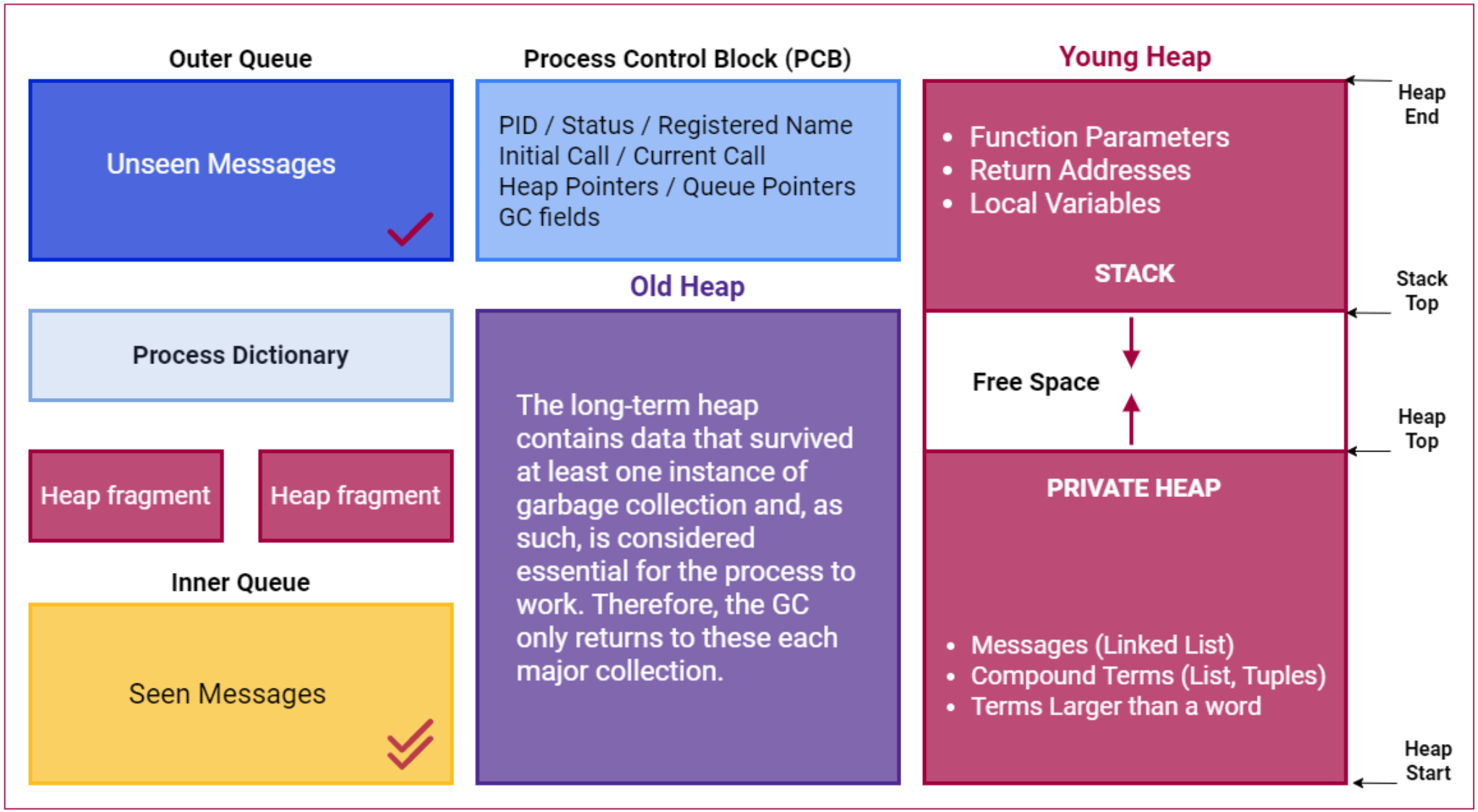
- Concurrency based on message passing and process model (Actor model).
- Process is isolated stage, no sharing memory, no deadlock, no “Global Interpreter Lock”.
- Many processes running independently, but not necessarily all at the same time.
- Parallelism is having processes running exactly at the same time.
- Individually garbage collection per process, thereby no “Stop the World”.
- Concurrency is supervised by implementations and pattern designs of OTP.
- Processes are controlled by preemptive scheduling mechanism.
- Processes' work loads are load balancing by schedulers and optimized hardware resources.

How concurrency really works

Welcome, astronaut!

Message passing ● Garbage Collection ● Scheduling

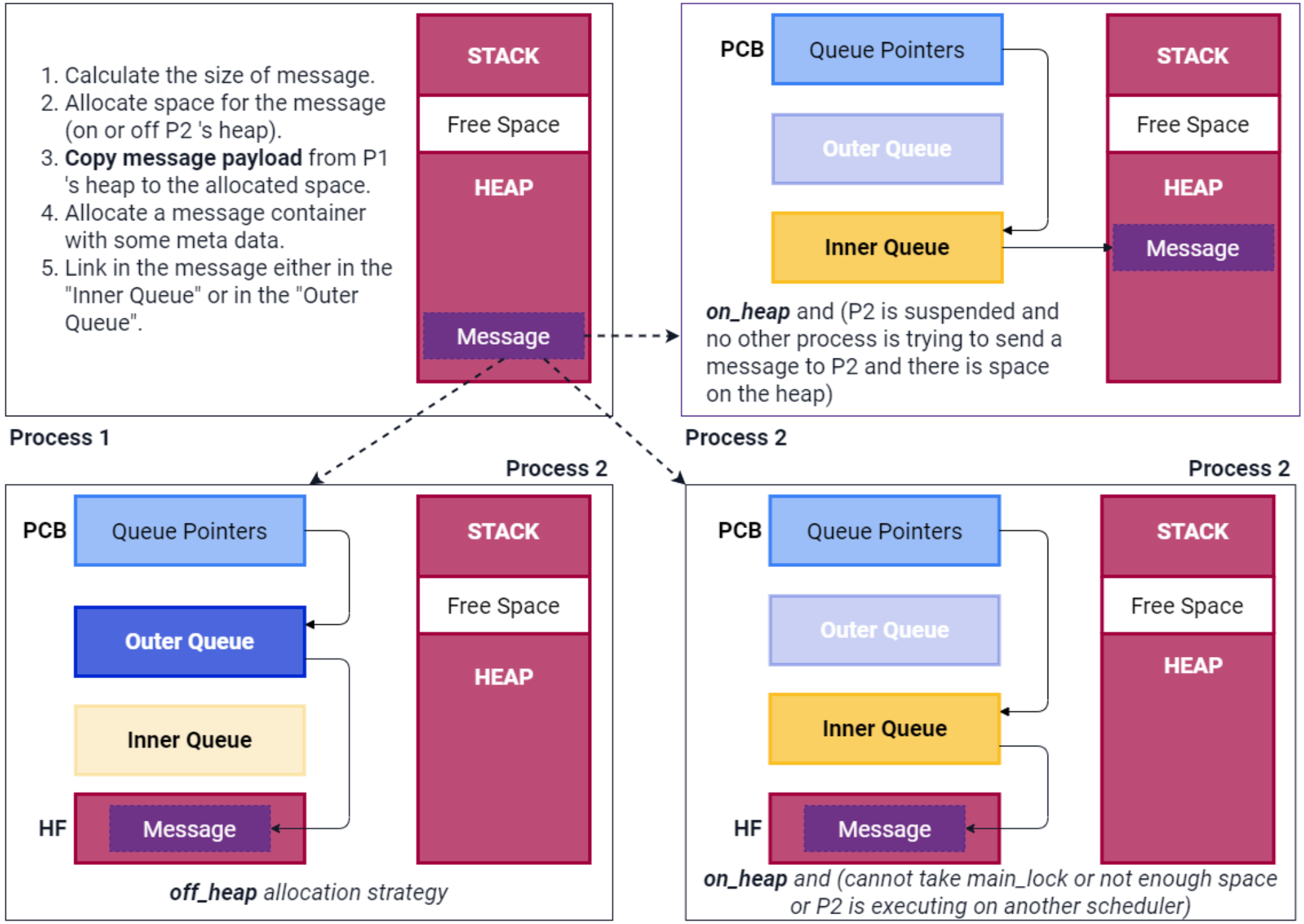
PROCESS MEMORY LAYOUT



Shared Area for Binaries

Shared Area for ETS Tables

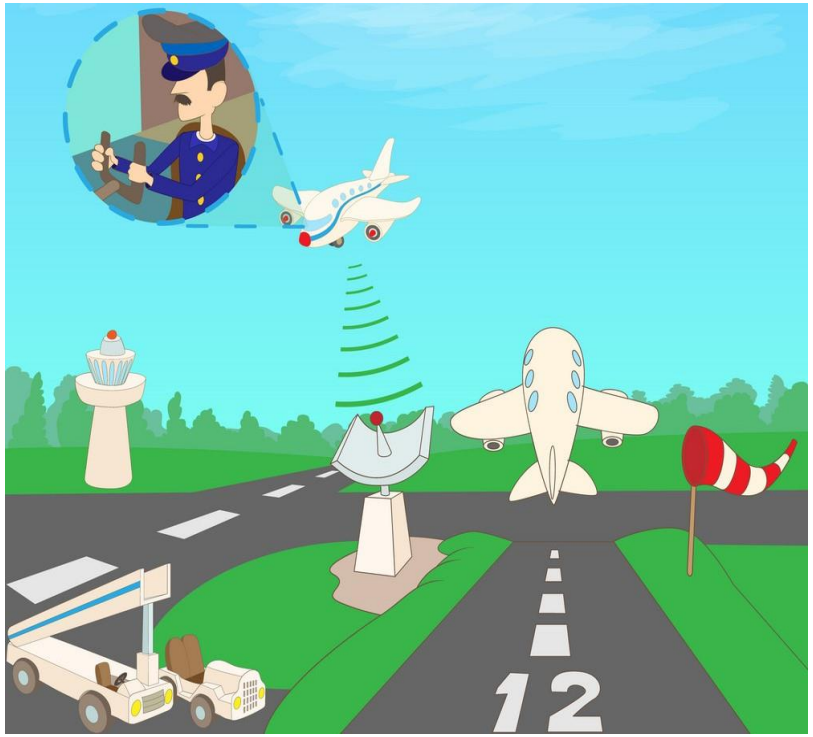
```
5> Pid = erlang:whereis(code_server).
<0.50.0>
6> erlang:process_info(Pid).
[{registered_name,code_server},
 {current_function,{code_server,loop,1}},
 {initial_call,{erlang,apply,2}},
 {status,waiting},
 {message_queue_len,0},
 {links,[<0.49.0>]},
 {dictionary,[]},
 {trap_exit,true},
 {error_handler,error_handler},
 {priority,normal},
 {group_leader,<0.46.0>},
 {total_heap_size,20317},
 {heap_size,2586},
 {stack_size,5},
 {reductions,99090},
 {garbage_collection,[{max_heap_size,#{error_logger => true,kill => true,
                                       size => 0}},
                      {min_bin_vheap_size,46422},
                      {min_heap_size,233},
                      {fullsweep_after,65535},
                      {minor_gcs,36}]},
 {suspending,[]}]
```


1. Calculate the size of message.
2. Allocate space for the message (on or off P2 's heap).
3. **Copy message payload** from P1 's heap to the allocated space.
4. Allocate a message container with some meta data.
5. Link in the message either in the "Inner Queue" or in the "Outer Queue".
STACK
Free Space
HEAP
Message
Process 1
PCB
Queue Pointers
Outer Queue
Inner Queue
STACK
Free Space
HEAP
Message
on_heap and (P2 is suspended and no other process is trying to send a message to P2 and there is space on the heap)
Process 2
Process 2
PCB
Queue Pointers
Outer Queue
Inner Queue
HF
Message
STACK
Free Space
HEAP
off_heap allocation strategy
Process 2
PCB
Queue Pointers
Outer Queue
Inner Queue
HF
Message
STACK
Free Space
HEAP
on_heap and (cannot take main_lock or not enough space or P2 is executing on another scheduler)

Messages Passing

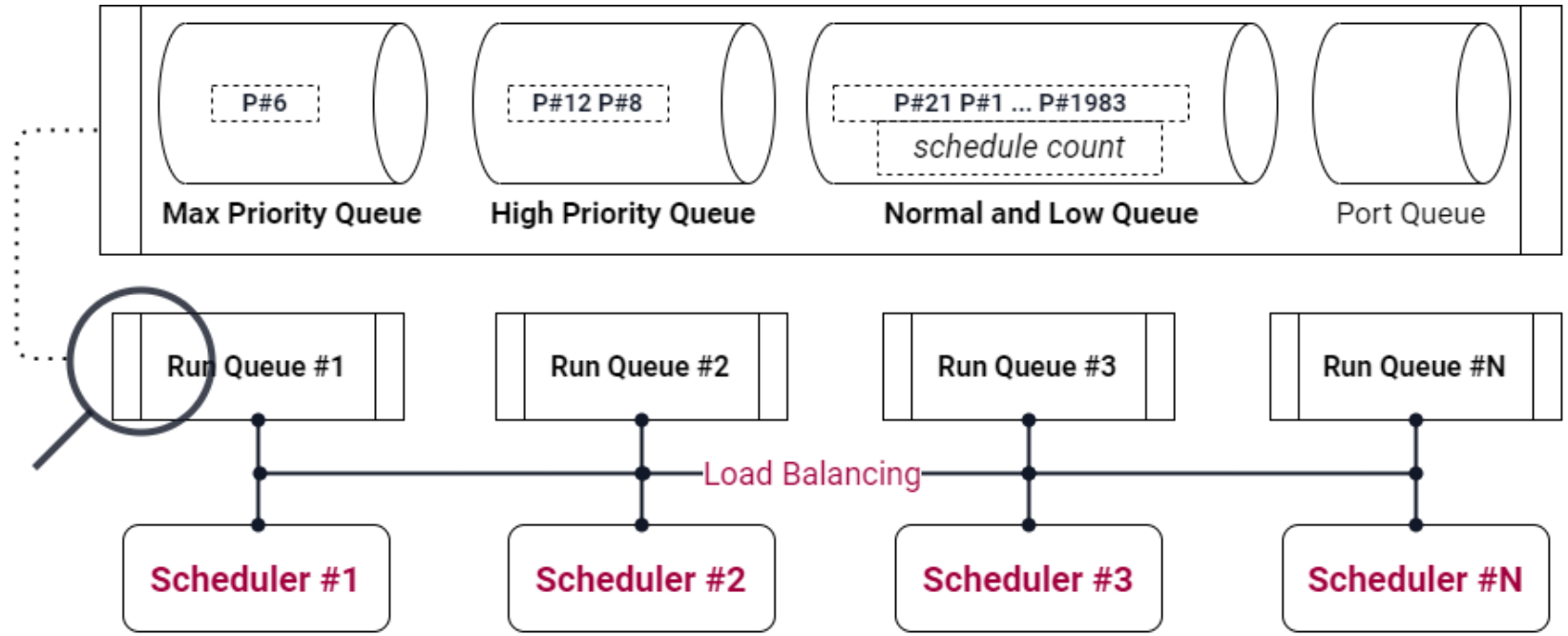
- Messages are asynchronous passed from the sender to receiver, that means sending a message doesn't require waiting for acknowledgement.
- If a process sends a message to itself, the message doesn't need to be copied. Only a new management data structure with a pointer to it is allocated.
- Messages between two processes are guaranteed to arrive in the order they were sent. In other words, if process A sends message 1 and then 2 to process B, message 1 is guaranteed to arrive before message 2.
- A process can see a message (in inner queue) is by matching them in a receive expression, the GC doesn't need to consider unmatched messages, therefore further improve performance.
- Using *off_heap* is a nice way for processes that receive a ton of messages as we get very little contention on the main locks, however, allocating a heap fragment is more expensive than allocating on the heap of the receiving process.
- In the upcoming release (*PR #5020*), Erlang/OPT will ship the optimization to parallelize sending message with the *message_queue_data=off_heap* setting enabled.

Cooperative vs Preemptive Scheduling



In *cooperative scheduling*, the running processes must cooperate for the scheduling scheme to work. The running processes proactively yield control periodically or when idle or logically blocked, then scheduler starts a new process and again waits for back voluntarily control.

In *preemptive scheduling*, the running processes could temporarily preempted (interrupt). This interrupt is done by an external scheduler by does context switching with no cooperation from the running processes. Context switching is done based on some factors like priority, time slice or reductions.



- The Run Queue is the FIFO queue that contents the pointers of *runnable* processes.
- There are three run queues for processes managed by a scheduler: One queue for *max priority* processes, one for *high* and one contains both *normal and low* processes.
- The default process priority is *normal,* and the *max priority* is reserved for internal use in Erlang runtime.
- Scheduler will pick the processes in *Max Queue* first, then *High Queue*. Until there are no processes in the *Max* and the *High* queues will the scheduler pick the first process from the *Normal and Low* queue. Processes in the same queue are executed in round-robin order.
- *Normal Process* has *schedule count* of 1, and *Low Process* has *schedule count* of 8. *Schedule count* is reduced by one when a process is picked from the queue, if the *count* reaches zero the process scheduled. This means that *Low priority processes* will retain (by re-enqueue) in the queue seven times before they are rescheduled.

References

- https://github.com/happi/theBeamBook
- https://beam-wisdoms.clau.se
- https://jlouisramblings.blogspot.com
- https://evrone.com/garbage-collection-erlang
- https://medium.com/@jlouis666
- https://evrone.com/garbage-collection-erlang
- https://www.erlang.org/docs
- https://elixir-lang.org
- https://github.com/erlang/otp
- https://youtu.be/LOfGJcVnvAk
- https://youtu.be/ArRr4trTCjQ
- https://learnyousomeerlang.com/
- Elixir IN ACTION Second Edition, Saša Juric
- Erlang AND OTP IN ACTION, Martin Logan - Eric Merritt - Richard Carlsson
- Characterizing the Scalability of Erlang VM on Many-core Processors, Jianrong Zhang
- Inside the Erlang VM with focus on SMP, Kenneth Lundin
- And others blog, media on internet.

iPlus

Thank you!

le_duc_tam

leductam