FOUR CONCURRENT RINGS

(Team work, teams of 1, 2, or 3)

1. OVERVIEW

Build a concurrent system composed of four independent rings of processes. You will implement this in Elixir, and then again in Java. In Elixir the rings will have spawned processes. In Java, these will be threads. Using the two programs, we will compare the relative performance of actor-style processes vs. threads on shared data.

A central Main process (Coordinator) will:

- Read integer inputs from the keyboard
- Route each integer to exactly one of four rings (based on value)
- Print a completion message from each ring when that ring finishes processing one input

Each ring:

- Contains *N* nodes arranged in a directed cycle (parameter *N* being given a value at start of execution)
- Circulates a work token (message) for exactly *H* hops (again, *H* being given a value at start)
- Allows at most one token "in flight" at a time (meaning one input is being processed in a ring at a time… other inputs needing that ring are waiting)
- Maintains FIFO ordering for queued inputs

The four rings may operate concurrently. So while the user may have input dozens of inputs, only 4 at most are being processed at any time

As noted, you will implement this twice, in Java (threads) and Elixir (processes/BEAM). ***You may use whatever AI coding tools and assistants you wish to, as much as you can, in order to create these working programs.***

2. PROGRAM PARAMETERS

At startup, the program receives:

N — ring size (number of nodes per ring), N ≥ 1

H — number of hops per token, H ≥ 1

For clarity, these parameters mean (for example) that if N is 1000 and H is 3000, each ring has 1000 processing nodes, and each input makes 3 full trips around the ring it is sent to.

If N is 50000, and H is 36, the each ring has 50,000 nodes, and each input visits the first 36 of those nodes (so the result is reported by node 36 and the other nodes are not visited). This is an unusual (but possible) combination of parameter values.

If N is 6000, and H is 12,456, the each work request goes twice around the ring and then a little more.

Of course, you will be trying your code with larger and large N values (100, 5000, 20000, 100000… etc.) until your machine cannot manage it. The parameter H can be used to keep your cores busy doing more arithmetic and less process swapping… like N of 5000 and H of 100000.

3. INPUT BEHAVIOR

The program repeatedly prompts the user for input.
If the user types "done":

- Stop accepting new input
- Finish all queued and in-flight work
- Shut down cleanly

Otherwise:

- Attempt to parse input as integer X
- If parsing fails, print an error
- If parsing succeeds, route X to a ring

Each accepted input is assigned a unique token_id.

4. ROUTING RULES

X < 0 → NEG ring
X == 0 → ZERO ring
X > 0 and even → POS_EVEN ring
X > 0 and odd → POS_ODD ring

5. RING STRUCTURE

Each ring consists of:

- Exactly N nodes (processes or threads)
- Nodes connected in a directed cycle
- A mechanism enforcing:
  - At most one token active in the ring at a time
  - FIFO queueing of additional inputs

The rings operate independently and concurrently.

6. TOKEN DEFINITION

Each token must contain:

token_id

ring_id
orig_input (constant)
current_val (mutable)
remaining_hops (initially H)

Initialization:
current_val := orig_input
remaining_hops := H

7. HOP PROCESSING

When a node receives a token:

1. Apply the ring-specific transformation to current_val
2. Decrement remaining_hops
3. If remaining_hops > 0, forward token to next node
4. If remaining_hops == 0, report completion to manager proc for reporting of results

All arithmetic must use signed 64-bit wraparound semantics.
(Java long overflow is acceptable; Elixir must implement wraparound explicitly.)

8. RING TRANSFORMATIONS

NEG: v := v * 3 + 1
ZERO: v := v + 7
POS_EVEN: v := v * 101
POS_ODD: v := v * 101 + 1

9. QUEUEING CONSTRAINT

For each ring independently:

- If the ring is idle, inject a new token immediately.
- If the ring is busy, enqueue the request in a FIFO queue.
- When the active token completes, inject the next queued request.

Invariant:
At most one token may circulate within a ring at any time.

10. CORRECTNESS REQUIREMENTS

Your implementation must:

- Route inputs correctly
- Execute exactly H hops per token
- Enforce FIFO per ring
- Enforce single in-flight token per ring
- Avoid deadlock
- Shut down cleanly after "done"
- Terminate all threads/processes properly

11. EXPLORATORY PERFORMANCE INVESTIGATION

You should instrument your system and then explore questions such as:

- How does increasing *N* affect latency?
- How does increasing *H* affect throughput?
- What happens if inputs are heavily skewed to one ring?
- Does Java behave differently from Elixir under load?
- How does the system scale as concurrency increases? Compare how many threads Java can deal with compared to how man processes Elixir can manage
- Where do you observe bottlenecks?
- Is performance CPU-bound? Synchronization-bound? Memory-bound?
- What scheduling behavior do you observe?

You may design and measure any performance metrics you find meaningful.

Here is some Elixir code that will create a monitor process that periodically (every 5 seconds) prints out some information about the processes and activities in the BEAM:

```
defmodule BeamMon do
  @moduledoc false

  def start(interval_ms \\ 5_000) do
    spawn(fn -> loop(interval_ms, true) end)
  end

  defp loop(interval_ms, first?) do
    receive do
      :stop ->
        :ok
    after
      if(first?, do: 0, else: interval_ms) ->
        mem_bytes  = :erlang.memory(:total)
        mem_gb     = mem_bytes / (1024 * 1024 * 1024)
        proc_count = :erlang.system_info(:process_count)
        run_queue  = :erlang.statistics(:run_queue)

        IO.puts("[BEAM] mem=#{fmt(mem_gb)} GB  procs=#{proc_count}
run_queue=#{run_queue}")

        loop(interval_ms, false)
    end
  end

  defp fmt(x), do: :io_lib.format("~.2f", [x]) |> List.to_string()
end
```

Add this to your Elixir code and run the start function in your code to run the monitor reporting process and watch how the beam is behaving as your code runs. Find a way to do something similar with Java about the JVM.

12. REQUIRED ANALYSIS

Submit a brief analysis (1–2 pages) discussing:

- What you chose to measure
- Why you chose those metrics
- What patterns you observed
- How Java threads compare to BEAM processes
- What surprised you if anything

Also explain what machine you are running these comparison experiments on, how many cores you have, how much RAM, and what trials you did (values of N and H),

13. DELIVERABLES

- Source code
- README file: explaining architecture and concurrency designs
- ANALYSIS file : Brief analysis document

14. Alternate Vesions

If you find this experiment intriguing, you can try an alternative or two.

The way it's been specified (with one token at a time in each ring), we are primarily seeing how many processes each system can create and manipulate, with only 4 (or a few more, like the managers) being active at a time. So the primary usage on the CPU cores is the 4 processes doing arithmetic, and then the BEAM and JVM doing the process/thread management… message passing, thread locking, waking up, etc.

Try lifting the "one active token per ring" restriction and letting (if you have sufficient inputs to process) all the nodes in a ring to be doing arithmetic concurrently.

Note that Elixir can manage this easily, as a mailbox is a FIFO queue. So you can just route all inputs intended for a ring to the first process in the ring… and the work requests will pile up waiting in the mailbox.

Java may require different thinking, depending on how your "one at a time" version is built. You might has built your "one at a time" version with a single token data structure that is locked by each process when it has a turn to work on it. To do more than one active computation at a time in a ring, you will need more than one token data structure and some way of managing them with appropriate synch.

Try varying the number of hops around the ring processes each input will do… like maybe when an input is routed to a ring, generate some number of hops randomly (maybe in some range, perhaps related to the H initial parameter… like some random integer multiple of H). Randomness this way will alter the repeatability of runs, as well as make the Elixir version give different results from the Java one for same inputs.

Maybe try having each node in a ring do something much more computationally intensive than just simple arithmetic. To keep repeatability, you might keep the same value transforms as in section 8 , then add some randomness by having an intensive “CPU burn” loop after that. Maybe compute this

Sq = sqrt(v); p = power(sq,sq); x = p; this is sqrt(v) raised to the power sqrt(v).

Convert this to int if necessary, and also make sure overflow works ok and doesn’t kill the computation. Maybe take sqrt of the sqrt to keep values smaller if needed.

Put this in a loop that runs some huge number of times (perhaps varying the number times randomly, or based on each hop). Adding the assignments will help make sure the compilers do not abstract the code out as “not doing anything”.

Just some thoughts. Have fun, do what you like.