Philosophers and Concurrency

Seema Fatima Bashir

23-02-24

1 Introduction

The assignment explores the concept of concurrency through the problem of dining philosophers. This report aims to stimulate the experience of five philosophers seated around a circular table, each with a bowl of noodles and only one chopstick between them. The challenge arises from the limited number of chopsticks which are divided amongst the philosophers, leading to potential deadlock in case all the philosophers wish to eat simultaneously. Thereby, this report will explore various solutions to this problem using Elixir, such that the philosophers don't starve. This will be done by managing the availability of chopsticks, and also experimenting with different scenarios to avoid potential challenges.

2 Chopsticks

A chopstick has two distinct states: :available, indicating that the chopstick is unclaimed, and :gone, indicating that it has been taken by a philosopher. Initially, a chopstick is in the :available state, awaiting a :request message. Upon receipt of this message, it responds with a :granted acknowledgment and transitions to the :gone state, signifying its occupation.Thereby, upon receiving a :return message, the chopstick reverts to the :available state, ready for another philosopher to claim it. This process ensures that the chopsticks are used smoothly by the dining philosophers, preventing any issues with sharing resources.

Thereby, a chopsticks module is created responsible for managing the availability and use of chopsticks. The start function initializes a chopstick process by spawning a linked process that begins in the available state. In the available state, the chopstick process waits to receive messages. If it receives a :request, from message indicating a philosopher's request for the chopstick, it responds by sending a :granted, self() message to the requesting philosopher and transitions to the gone state where it waits for a :return message to return the chopstick to an available state. If a :quit message is received, the process exits.

```
def start do
   stick = spawn_link(fn -> available() end)
```

```
end

def available() do
  receive do
    {:request, from} ->
      send(from, {:granted, self()})
      gone()
    :quit -> Process.exit(self(), :kill)
  end
end
```

Thereby,a chopstick can also be requested by a philosopher and can only be taken in case a granted is received. The request function is this defined to be used when only one chopstick is needed. It sends a :request, self() message to the specified chopstick process and waits for a :granted, _ message. If the timeout is reached before receiving a response, it returns :timeout.

Furthermore, the return function sends a :return message to the specified chopstick process, indicating that it is returning the chopstick to the available state. Lastly, the quit function sends a :quit message to the specified chopstick process, causing it to exit.

```
def return(stick) do send(stick, :return) end
def quit(stick) do send(stick, :quit) end
```

3 A philosopher

A philosopher can be in one of two states: dreaming or eating. When in the dreaming state, the philosopher is essentially inactive, waiting for the opportunity to eat. To begin eating, the philosopher must first request the left chopstick, followed by the right chopstick. Once both chopsticks are successfully acquired, the philosopher initiates the eating process.

Within the philosophers module, the implementation of the dreaming and eating processes involves the use of the :timer.sleep/1 library function. This function allows the program to pause execution for a specified number of milliseconds before continuing. Additionally to achieve randomness, the :rand.uniform is also utilized.

The module includes a start/5 function designed to spawn a philosopher process using spawn_link/1. This function requires five arguments: hunger, indicating the number of times the philosopher should eat before sending a :done message to the controller process; right and left, the process identifiers of the two chopsticks; name, a string representing the philosopher's name for logging purposes; and ctrl, the controller process that should be notified when the philosopher finishes eatings.

3.1 Successfully receiving a chopstick

Now, the philosophers module also implements other functions which allow for better management, as the philosophers eat. One function which is implemented is the make_order function.

```
def make_order(waiter, pos) do
    send(waiter, {:order, self(), pos})
    receive do
      :order_taken -> :ok
      :unavailable -> :no
    end
  end
```

The make_order function orchestrates the philosopher's structured request for chopsticks from the waiter. It takes two essential parameters: "waiter," representing the waiter's specific process identifier, and "pos," denoting the philosopher's position at the dining table. Initially, the philosopher dispatches an order message to the waiter, containing vital details like the order type (:order), the philosopher's unique process identifier (self()), and their position (pos). Subsequently, the philosopher enters a waiting state, awaiting a response from the waiter. Upon receiving a response, two potential scenarios unfold: If the response is :order_taken, it signifies the successful processing of the order by the waiter, enabling the philosopher to proceed with dining. Conversely, if the response is :unavailable, it indicates that the waiter is presently unable to fulfill the order, prompting the philosopher to either retry the request later or address the situation accordingly. Depending on the received response, make_order returns either :ok to confirm successful order placement or :no to indicate that immediate processing of the order was not feasible.

3.2 The Dreaming and then Eating

Additionally, the dream function is created to control the philosophers actions during the dining. Initially, it notifies the controller process (ctrl) of the philosopher's dining initiation by sending the :done message. Next, it pauses time using the sleep function, this is to represent the dreaming. Thereby, the philosopher invokes the make_order function to request chopsticks from the waiter (waiter) at its designated position (pos).

Upon receiving a response, the dream function assesses the outcome using a case statement. If the order is successfully placed (:ok), the philosopher proceeds to acquire both the left and right chopsticks through the Chopstick.request function. If successful, a log message confirming that it has received both chopsticks is diplayed. The philosopher then simulates the eating process by pausing before recursively calling itself with updated parameters to continue dining.

```
// Section of code from make_order function

 case make_order( pos) do
      :ok -> # with waiter
        case Chopstick.request(left, right) do
          :ok ->
            IO.puts("#{name} received both chopsticks")
            sleep(1000)
            eat(hunger, right, left, name, ctrl, pos, backoff + 1800)
```

4 Dinner at the table

To initiate the philosophers and manage their execution, a third file named ”Dinner” is implemented. Within this module, all processes, including the philosophers and the chopstick processes, are started under a controlling process. This controlling process oversees the activities of the philosophers and ensures the proper termination of the chopstick processes upon completion of the program. Additionally, the controlling process handles error scenarios and potential program errors by sending an :abort signal, which exits the program.

```
Philosopher.start(5, c1, c2, "Arendt", ctrl, timeout, waiter, 0, 0, rem(seed, 82))
Philosopher.start(5, c2, c3, "Hypatia", ctrl, timeout, waiter, 1, 0, rem(seed * 2, 51))
Philosopher.start(5, c3, c4, "Simone", ctrl, timeout, waiter, 2, 0, rem(seed * 3, 321))
....
```

4.1 Deadlocks

Moving one, a potential deadlocks can arise when philosophers are unable to acquire the necessary chopsticks due to circular dependencies. For instance, if philosopher A waits for chopstick B, B waits for chopstick C, and C waits for chopstick A, a deadlock occurs. To address this issue, we've introduced a timeout mechanism. If a philosopher waits for a chopstick beyond a specified duration, we terminate the request and return the philosopher to a dreaming state, allowing them to retry later. Another deadlock scenario involves a philosopher starving, where repeated timeouts prevent them from eating altogether.

To handle deadlock situations, the module introduces the wait function, which serves as a monitoring mechanism for the philosophers' activities and the availability of chopsticks. The wait function operates recursively, continuously observing the philosophers' interactions and chopstick usage.

Initially, the wait function is invoked with parameters indicating the number of iterations remaining (n), the list of chopsticks, and the start time of program execution. Within the wait function, a receive block listens for messages indicating various events.

If the received message is :done, indicating that a philosopher has completed a dining cycle, the wait function decrements the iteration count (n) and continues to monitor the dining process. In the case of an :abort message, signaling a potential deadlock or error, the wait function terminates the program by exiting with a kill signal.

```
def wait(0, chopsticks) do
  Enum.each(chopsticks, fn(c) -> Chopstick.quit(c) end)
end
def wait(n, chopsticks) do
receive do
    :done ->
      wait(n - 1, chopsticks)
    :abort ->
      Process.exit(self(), :kill)
end end
```

4.2 Sequential vs Asynchronous

Up until now, Processes have been executed sequentially, meaning that a philosopher would wait for the first chopstick to be granted before requesting the second one. Thereby an asynchronous chopstick request is introduced. Unlike the previous methods, where philosophers waited for a single chopstick, this approach involves sending requests to both chopsticks concurrently and then awaiting responses. This adjustment aims to enhance execution speed and optimize the handling of chopstick requests.

```
     def request(left, right, timeout) do
        send(left, {:request, self()})
        send(right, {:request, self()})
        receive do
        {:granted, stick} ->
      IO.puts("1st chopstick taken"); granted(stick, timeout)
        after
            timeout -> :timeout
      end
    end
```

The request function is used in the dining philosophers problem to ask for chopsticks. It takes three inputs: left and right, which are the identifiers of the left and right chopsticks respectively, and timeout, which is the maximum time allowed for a response. This function sends a request message to both chopsticks and waits for a response. If both chopsticks respond within the specified time, indicating that they've been granted, the function prints a message saying the first chopstick is taken and proceeds with further actions. But if there's no response within the timeout period, it returns a timeout signal.

Additionally, the "granted" function is created in order to manage the process of acquiring the second chopstick for a philosopher. It employs a receive do to listen to a :granted, _ message. In that case, the second chopstick is acquired successfully. Thereby, the function also prints an :ok message to the console confirming that the chopstick has been acquired.

On the other hand, if no message is received within the specified timeout, the function triggers the after clause, which invokes the return(stick) function to return the first chopstick. This allows for the chopsticks to be used correctly and prevents situations where the philosophers can't get both chopsticks, and thereby are stucks.

```
def granted(stick, timeout) do
  receive do
    {:granted, _} -> IO.puts("2nd chopstick taken"); :ok
  after
    timeout ->
      return(stick)
      :timeout
  end
end
```

5 Benchmarking

Table 1: Benchmarks for syncronous and asynchrnous implementation.

Algorithm	With timeout	n	Times eaten	Exec time (s)	Eats every nth (s)
Sequential	No	100	95	40	0.2
Sequential	Yes	100	90	28	0.3
Asynchronous	No	100	95	41	0.4 - 0.6
Asynchronous	Yes	100	Starved	Starved	Starved
Asynchronous	Yes	100	70	32	0.5
Asynchronous	Yes	50	Starved	Starved	Starved
Asynchronous	Yes	50	30	9	0.3

Each algorithm is assessed based on whether it employs timeouts, the number of philosophers involved (n), the number of successful dining completions, the execution time in seconds, and the frequency of dining. The sequential algorithm, without timeouts, demonstrates a high success rate among philosophers, with consistent execution times and dining frequencies. However, when timeouts are introduced, the success rate slightly decreases, and the execution times become more variable. On the other hand, the asynchronous algorithm, while effective in scenarios without timeouts, experiences challenges when timeouts are enabled, leading to starvation among philosophers and longer execution times.

6 Waiters

To optimize the dining process and prevent deadlocks among philosophers, a more effective strategy involves introducing a waiter that manages the allocation of resources. One proposed solution is to implement a waiter with three distinct states: "take order," "take second order," and "unavailable." In this model, philosophers request to eat by interacting with the waiter, who coordinates the process to ensure that only a limited number of philosophers can attempt to eat simultaneously. By controlling the dining process in this manner, the waiter decreases the risk of deadlocks, as philosophers are guided through a structured sequence of actions. The waiter's intelligence lies in its ability to dynamically adjust the availability of dining opportunities based on the current state of the dining table, ensuring that the maximum number of philosophers can attempt to eat without encountering deadlocks.

Thereby, the take_orders function is created.The initial state of the waiter, where it waits to receive orders from philosophers. Upon receiving an order message, it acknowledges receipt by sending an :order_taken message back to the philosopher and then proceeds to the take_second_order function to process the order further. In the take_second_order function, the waiter receives additional instructions from philosophers, either to return a chopstick or to place a new order. If a return message is received, the waiter transitions back to the take_orders state to await further instructions. Otherwise, if a new order message is received, the waiter determines the availability of adjacent chopsticks and responds accordingly, either marking them as unavailable or proceeding to fulfill the order. This iterative process ensures orderly coordination between philosophers and the waiter in managing dining resources.

```
 def waiter_unavailable(occupied1, occupied2) do
    IO.puts("Waiter is unavailable")
    receive do
      {:return, pos} ->
        if occupied1 == pos do
          take_second_order(occupied2)
else
          take_second_order(occupied1)
        end
      {:order, from, _} ->
        send(from, :unavailable)
        waiter_unavailable(occupied1, occupied2)
end end
```

Another function created is called waiter unavailable.The waiter_unavailable function represents a scenario where the waiter is unable to fulfill a philosopher's order due to chopstick unavailability. Upon entering this state, the waiter notifies observers that it is unavailable. It then awaits instructions from philosophers, expecting either a request to return a chopstick or a new order request. If a return message is received, indicating that a philosopher has returned a chopstick, the function redirects the flow to take_second_order, enabling the waiter to process subsequent orders. However, if a new order message is received while chopsticks are still occupied, the waiter responds by marking them as unavailable and continues to wait for further instructions

```
def non_adjacent(pos) do {rem((pos + 2), 5), rem((pos + 3), 5)} end
```

There are an equal number of philosophers and chopsticks, only two philosophers can eat simultaneously. To manage this, the waiter has two operational states for taking orders, transitioning to an unavailable state during order processing. As mentioned earlier, a philosopher can only eat when their adjacent neighbors are dreaming, highlighting the need for coordinated serving. The non_adjacent function calculates the positions of the philosophers who are not adjacent to the given position, determining which philosophers can be served next based on the first order taken.

6.1 Waiter Benchmark

In an effort to assess the waiter's effectiveness, a new parameter called "back-off" is introduced to regulate the duration that a philosopher waits before attempting to acquire chopsticks again after an initial failure. Unfortunately, the introduction of the waiter resulted in an increased number of timeouts, leading to starvation among some processes. Only when employing very high back-off times, the program able to execute successfully.

Table 2: Benchmark for waited implementation

Back-off	With timeout	n	Times eaten	Exec time (s)	Eats every nth (s)
0	Both	100	Starved	Starved	Starved
400	Both	100	Starved	Starved	Starved
800	Both	100	Starved	Starved	Starved
1600	Both	100	Starved	Starved	Starved
2000	Both	100	Starved	Starved	Starved
2000	Yes	100	55	75	0.8