Task 11 - Philosophers and Concurrency

Casper Kristiansson

March 1, 2022

Introduction

Task eleven of the course Programming II consisted of implementing a concurrency problem in elixir. The task basic principle involves managing different processes which has overlapping resources. For example, in this problem five different processes needs two items to work (chopsticks), while all the processes together only share a total of five items (chopsticks). This means that the different processes need to share resources and wait for their turn before they can perform a certain action.

Result

The task can be divided into three different sections, one for controlling and creating sub-processes (chopsticks), one for managing and creating the main sub-processes (philosophers) and lastly a solution for managing deadlocks.

Managing Chopsticks

The first part of the task was to oversee the chopstick processes. To create a process in elixir the author uses the function **spawn_link** which receives a function as a parameter. The program than needs two different function to manage if a chopstick is available or gone. The method uses the receive statement in elixir which will wait until it receives a message. When it does, the process (chopstick) is being used by a philosopher. Then the method sends a message to the processes that the philosopher is granted access to it. The method also initiate the gone method which will wait until the chopstick receives another message.

```
def start do
  stick = spawn_link(fn -> available() end)
  {:id, stick}
end
def available() do
  receive do
    {:request, from} ->
      send(from, :granted)
      gone()
    :quit ->
      :ok
  end
end
```

Figure 1: Function for creating a new process and checking if the process is available

The program than uses a **request** function which receives a stick as parameter and waits until a given chopstick is available by checking if the return message is *:granted*. This is the function which could cause the program to receive deadlocks, because it will wait until it receives a message which is when the chopstick is available and not being used.

```
def request({:id, stick}) do
  send(stick, {:request, self()})
  receive do
    :granted ->
      :ok
  end
end
```

Figure 2: Sending a request to receive a chopstick

The last part of this subtask is to create a function for handling if a chopstick should be released and if it should be terminated. These functions are implemented by simply passing a message to the process.

```
def quit({:id, stick}) do
  send(stick, :quit)
end
def release({:id, stick}) do
  send(stick, :relase)
end
```

Figure 3: Terminating and returning a process

Managing Philosophers

The next part of the task is to manage the different philosophers. The module contains different functions for handling the philosophers different states *dream, eat, done*. The program starts of by creating a process which receives two different chopstick process ids, hunger level, name, and a controller. The program starts of by spawning a new process for the function dream. The dream function will than wait a random amount of time before calling the function eat.

```
def start(hunger, right, left, name, ctrl) do
  spawn_link(fn -> dream(hunger, right, left, name, ctrl) end)
end

def dream(0, _, _, name, ctrl) do
  send(ctrl, :done)
end
def dream(hunger, right, left, name, ctrl) do
  sleep(200)
  eat(hunger, right, left, name, ctrl)
end
```

Figure 4: Creating and managing the dream state

The next part is to implement the function to oversee the state for eating. The process main goal is to receive the correct chopsticks before the philosopher can eat. The processes do this by simply sending a request to the Chopstick module and waits until the desired chopstick is available. This is the second part of the program which causes the deadlocks.

```
def eat(hunger, right, left, name, ctrl) do
  case Chopstick.request(right) do
    :ok ->
      case Chopstick.request(left) do
        :ok ->
          done(hunger, right, left, name, ctrl)
      end
  end
end
```

Figure 5: Requesting chopstick processes

The next part of this task is to implement the module for managing the situation where the philosophers is done eating. The function simply starts of by releasing the chopsticks and then calling the dream function again with one less hunger. The function than will recursively do this until the dream function receives a hunger of zero, which means that the philosopher is done eating.

```
def done(hunger, right, left, name, ctrl) do
  Chopstick.release(right)
  Chopstick.release(left)
  dream(hunger - 1, right, left, name, ctrl)
end
```

Figure 6: Releasing processes and calling the next recursion loop

Deadlooks

The current solution of the problem could cause deadlocks where the different philosophers could be waiting for processes after each other which means that the program will get stuck. A way to manage this is to create a maximum timeout which should represent how long a philosopher is willing to wait for a chopstick. This can easily be solved by using the term **after** in elixir. The term will cause the program to wait a maximum of amount of time for a message to arrive and if it doesn't it will return (:timeouted)

```
def request({:id, stick}, timeout) do
  send(stick, {:request, self()})
  receive do
    :granted ->
      :ok
    after timeout ->
      :timeouted
  end
end
```

Figure 7: Maximum time the process is willing to wait for a message, (chopstick is available)

Then the only thing that is required is to update the case statement which should manage what the function should do if the process has achieved the timeout. To keep it simple the function just waits a bit of time and then try to request the chopstick again.

```
def eat(hunger, right, left, name, ctrl) do
  case Chopstick.request(right, 300) do
    :ok ->
      case Chopstick.request(left, 300) do
        :ok ->
          done(hunger, right, left, name, ctrl)
        :timeouted ->
          Chopstick.release(right)
          dream(hunger, right, left, name, ctrl)
      end
    :timeouted ->
      dream(hunger, right, left, name, ctrl)
  end
end
```

Figure 8: Dealing with deadlooks

The author also provided specific seed value to each philosopher to minimize the random behavior of the program. The seed that was chosen was five hundred with an offset of one for each process. The program will than with the help of the command *:rand.seed(:exsss, seed, seed, seed)* give the exact same sleep pattern for the processe. This can be extremely useful for debugging and fixing where certain situations the program could cause deadlocks.

Discussion

The author thought that dealing and working with concurrency in elixir was extremely fun. But the author did have a bit of problem implementing and dealing with the philosopher's module where it constantly would get stuck due to deadlocks. Therefore the author needed to use **IO.puts** commands to show what different states the different philosophers where in. From that point the author was able to backtrack and fix what was wrong.

The author ran a couple benchmarks on the specific seed mentioned above. He evaluated a range of different hunger sizes and checked how long each runtime was and how the sleep timer before eating affected the overall time. The author found that the best configuration was that the process waited 100us before eating with timeout of 300us (maximum of time waiting for a chopstick). With this setup the processes would receive a total of three timeouts. If the sleep timer before eating would be increased to around 130-150us the program wouldn't receive any timeouts at all. But even though the program received zero timeouts the previous configuration was still faster having a lower runtime.

A way to improve this is both two control how many philosophers is allowed to eat the same time using a (waiter). The easiest way is to create a simple check how many processes is currently eating. But a better way would check if a process that share the same resources is currently eating. When that process is done, the new process could try to start eating again. Using this strategy, the program would be using more of a queue system rather than just brute forcing until everyone is done eating.

Another thing that could be improved is by implementing a dynamic sleep timer. If a process is timeout from waiting to long for a chopstick the sleep timer is increased (timeout time could also be increased). By doing this the timer will reach appoint where the philosophers never get timeout. In a situation where each philosopher would have a hunger of one hundred the configuration would really benefit from having a dynamic timer.

In theory even with combining all the different solutions the program could theoretically still get stuck in a deadlock. But with all the precautions the program is very unlikely to end up in a deadlock. In some cases, the program could get stuck with two remaining processes but usually after a bit of time the processes is able to eat and finish up the program.