Philosophers and Concurrency

Axel Månson Lokrantz

Spring Term 2023

Introduction

The dining philosophers is a famous problem in computer science. The problem consist of the following statement: Five philosophers sit around a table. All philosophers have a plate of food and a chopstick. Each philosopher has two states, dream where the philosopher does nothing for a set amount of time or eating. To be able to eat the philosopher needs two chopsticks, meaning both of the adjacent philosophers has to be in a state of dreaming. After an individual philosopher finishes eating they will put down both their chopsticks. The problem is to design an algorithm such that no philosopher will starve. Meaning all processes can continue alternate between eating and idle forever. The solution was written in Elixir together with Daniel Dahlberg at KTH.

Github link

A link to the program in its entirety can be seen here.

A Chopstick

A chopstick is represented by a process and can either be `:avaliable` if the chopstick is available or `:gone` if the chopstick is taken. A chopstick starts in a state of `:avaliable`, waits for a `:request` and returns `:granted`. The chopstick now moves to its state of being `:gone` until it receives `:return` at which the state of the chopstick changes back to being `:avaliable`.

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

```
  def gone() do
    receive do
      :return -> available()
      :quit -> Process.exit(self(), :kill)
    end
  end

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

A chopstick is requested by a philosopher and only taken if it receives `:granted` at which the request function returns an `:ok` to the philosopher. Note that there is a timeout in the implementation, which will be covered later in the report.

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

A Philosopher

A philosopher can either be in a state of dreaming or eating, everything in between the philosopher is waiting for a chopstick. In the dreaming state the philosopher dreams, meaning she does basically nothing. To be able to eat, the philosopher first has to request the left chopstick before she can request the right chopstick. When both chopsticks are acquired the philosopher begins to eat. To get some variation to our program we decided to implement a sleep function that randomizes for how long a philosopher eats and dreams. To start a philosopher process we provide the following arguments:

- `hunger:` The current hunger of a philosopher. When a philosopher eats the value will increase by 1. If the philosopher requests chopsticks

but receives a `:timeout` hunger will decrease by 1. If hunger reaches zero the philosopher dies of starvation and the program stops.

- `right` and `left:` the process identifiers of the two chopsticks.
- `name:` The name of the philosopher to tell them apart.
- `ctrl:` The controlling process that should be informed when the philosopher is done.

Dinner at the table

To start the philosophers we created a third file called `Dinner.ex`. All processes are started under a controlling process that keeps track of all the philosophers and makes sure that the chopstick processes are terminated when we are done. If an error occur or if the program is stuck we send an `:abort` which exits the program.

```
Philosopher.start(5, c1, c2, "A", ctrl, timeout, 0, rem(seed, 82))
Philosopher.start(5, c2, c3, "B", ctrl, timeout, 1, rem(seed * 2, 51))
Philosopher.start(5, c3, c4, "C", ctrl, timeout, 2, rem(seed * 3, 321))
...
```

Deadlock

With the above implementation our program runs fast but is susceptible to deadlocks. A deadlock can occur if philosopher A waits for B, B waits for C and C waits for A for example. To break the deadlock we introduced a timeout. If a philosopher waits for a chopstick for a set amount of time we terminate the request and put the philosopher back to a dream state. Another type of deadlock situation is if a given philosopher starves, meaning she gets repeated timeouts and never gets to eat.

Sequential vs Asynchronous

Up until now all the processes has been done in a sequential order, meaning a given philosopher waits for the first chopstick to be granted before requesting the second chopstick. For benchmark purposes we created an asynchronous request for chopsticks. Instead of waiting for a given chopstick this implementation sends a request to both chopsticks and then waits for a reply.

```
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

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

Below the benchmarks for the two implementations can be seen.

Algorithm	With timeout	n	Times eaten	Exec time (s)	Eats every nth (s)
Sequential	No	100	95	30	0.3
Sequential	Yes	100	90	23	0.3
Asynchronous	No	100	95	41	0.4 - 0.6
Asynchronous	Yes	100	Starved	Starved	Starved
Asynchronous	Yes	100	60	34	0.5
Asynchronous	Yes	50	Starved	Starved	Starved
Asynchronous	Yes	20	15	7	0.5

Table 1: Benchmarks for syncronous and asynchrnous implementation.

The waiter

A better strategy would be to introduce a waiter that controls how many philosophers that can eat at any given time. We had several ideas and ended up with a waiter that has three states `take_order`, `take_second_order` and `unavailable`.

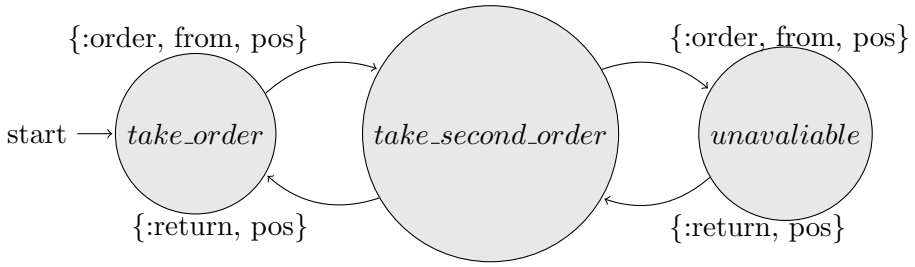
```
def take_orders() do
  IO.puts("Waiter is taking orders")
  receive do
    {:order, from, pos} ->
      send(from, :order_taken)
```

```
      take_second_order(pos)
  end
end

def take_second_order(occupied) do
  IO.puts("Waiter is taking second order")
  receive do
    {:return, _} -> take_orders()
    {:order, from, pos} ->
      {phil1, phil2} = non_adjacent(occupied)
      if pos == phil1 or pos == phil2 do
        send(from, :order_taken)
        waiter_unavailable(occupied, pos)
      else
        send(from, :unavailable)
        take_second_order(occupied)
      end
  end
end

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
```

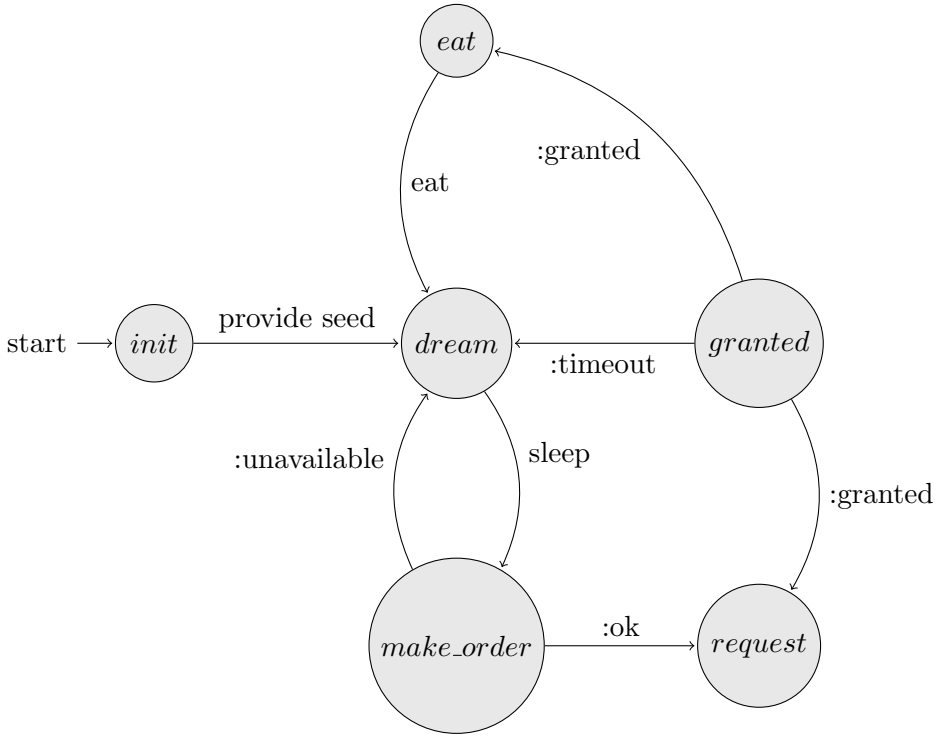
Since there are as many philosophers as there are chopsticks only two philosophers can eat at the same time. Therefor the waiter has two states where she is able to take orders, while processing these orders she goes into a state of unavailable. As previously mentioned, a philosopher can only eat while their adjacent neighbours dream. Therefor we created a function `not_adjacent` that calculates which philosophers to serve next based on the philosopher who took the first order.

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

State diagram: Waiter



State diagram: Philosopher



The state diagram for the philosopher is with the waiter implementation. A state diagram for the chopstick has been left out in this report, but can be seen in the assignment pdf.

Waiter benchmark

To benchmark the waiter we introduced a new argument "back-off", so that a philosopher will wait a set amount of time before trying to grab the chopsticks after a failed attempt. Unfortunately, with the waiter our program got too many timeouts leading to starved processes. It was only for really high back-off times (2 seconds) the program could compile. We later discovered that this could be due to the fact that every time a philosopher requests

chopstick and gets rejected we send a timeout, instead of just letting the philosophers stay in line. Another idea we had to improve the algorithm was to make the back-off time dynamic, so that the time would increase if a process was closed to being starved, and decreased if the processes were healthy. With a little more time at our disposal and more room to experiment the results would probably differ significantly.

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	65	85	0.7

Table 2: Benchmark for waited implementation.