Five cats

Praanto Samadder

February 23, 2024

1 Meet our philosophers

Figure 1: Meet our thinkers

Our philosophers today are famous internet cats.

Also, instead of implementing our algorithm decentralised first, and then with a centralised 'waiter', I did the other way.

Furthermore, I have flipped the terminologies a bit. What is referred to as strength in the problem PDF is referred to as hunger (as in time until the cat succumbs to hunger) and what is referred to as hunger in the problem PDF is referred here as food. Hunger is replenished every time a cat gets given a pair of chopsticks.

2 Cats as processes

Each of our cats can be it's own separate process, along with a 'Katnel' (aka the waiter) as a separate process. Messages are passed cat-to-waiter and waiter-to-cat. Cats do not talk to each other.

3 The `Katnel`

The `Katnel` (which works like a kernel) works like a kernel. The `Katnel` handles chopstick requests and returns. The cats, when sending a message, also send `self()` which is a reference to their own process ID to allow the kernel to send a message back.

3.1 Handling messages

The `Katnel` handles two types of messages: chopstick requests and chopstick returns. Cats send messages for chopsticks in pair, although they are kept track of individually.

We make use of two maps: `requestMap` and `availabilityMap`. Request map is a map storing all the requests `Katnel` has received from the cats. We first store it in our queue before we pick the request with the lowest hunger (i.e. highest chance of dying) to serve first. We use the `handleRequest/2` to to add to our stack and `serveRequest/2` to send a message to the appropriate process. The cats, within their message also send their own PIDs (as `self(())`) which allows the kernel to send responses.

Once a request has been dealt with, the request is removed from the queue. This may risk the chance of the cat dying however due to the nature of the algorithm, that particular cat will soon be sending back another request that we can serve in the next round of calls.

The `availabilityMap` is a map of all the chopsticks and their availability. Each chopstick can either be (:taken) or `:available`, denoted by an atom. We make use of the `checkChopstickAvailability/3` method to determine if we can even approve this request.

```
1  def run(requestMap, availabilityMap) do
2    receive do
3      {:request, cat, hunger, c1, c2, cattoPid} ->
4        requestMap = handleRequests(requestMap, cat, hunger, c1, c2,
       cattoPid)
5        serveRequest(requestMap, availabilityMap)
6
7      {:return, c1, c2} ->
8        a = handleReturns(availabilityMap, c1, c2)
9        serveRequest(requestMap, a)
10   end
11 end
```

How message handling is implemented

3.2 Keeping track of the chopsticks

The chopsticks can be kept track of by using a chopstick map. When a cat requests to borrow a pair of chopsticks, the `Katnel` checks if those particular chopstick(s) are `:available`. If they are, well, life's good. The chopsticks are borrowed. If the chopsticks are `:taken` however, well sucks to be you I guess.

```
def checkChopstickPairAvailability(map, c1, c2) do
  case Map.get(map, c1) do
    :taken -> :sad_face

    :available ->
      case Map.get(map, c2) do
        :taken -> :sad_face
        :available -> :ok
      end
  end
end
```

Check if a chopstick pair is available for request

3.3 Dealing with request messages

When a cat sends a message requesting a chopstick, their request is first placed in a min-priority queue. After it has been placed there, the cat with the lowest hunger value is served first. This hunger value is the same value as the *strength* value in the problem PDF.If the hunger value of a cat reaches 0, well, the cat dies.

```
# place in queue
def handleRequests(rqMap, cat, hunger, c1, c2, cattoPid) do
  Map.put(rqMap, cat, {hunger, cattoPid, c1, c2})
end

# serve lowest hunger
def serveRequest(updatedMap, csaMap) do
  {{key, {_, cattoPid, c1, c2}}, map} = getLowestIn(updatedMap)
  case checkChopstickPairAvailability(csaMap, c1, c2) do
    :sad_face ->
      send(cattoPid, :denied)
      run(map, csaMap)
    :ok ->
      csaMap = Map.put(csaMap, c1, :taken) |> Map.put(c2, :taken)
      send(cattoPid, {:approved, c1, c2})
      run(map, csaMap)
  end
end
```

Serving request with the lowest hunger value

Determining the lowest hunger value in a map is trivial when using the `Enum.min/2` method.

When a cat sends a message to return a pair of chopsticks, the chopsticks map is updated making them now available for requests.

4 Cats

How do our ~~philosophers~~ cats maintain a work-life balance? Through message passing of course!

Each cat has a food value (how much they need to eat before they are full), a hunger value (when hunger becomes 0, they die), and a laziness value (how long they ~~sleep~~ think before they must eat again).

A cat makes a request to the kernel and awaits a response (via the thread-blocking `receive` method). If the request is `:denied`, hunger is decremented and `meow/8` is called again to form a recursive loop. If a request is `:approved` however, the cat spends some time eating (for 1 millisecond for our demonstration), then spends some time sleeping before requesting the `Katnel` again for chopsticks.

Notable that the `receive` block is thread-blocking so our cat does nothing while it waits for an answer from the server. Maybe it should've spend some time doing some thinking while it waited.

Finally when the food value reaches 0, the cat is done eating and can notify the kernel about it's termination.

```elixir
def meow(cat, krnlPID, c1, c2, food, base_hngr, hngr, laziness) do
  # request chopsticks
  send(krnlPID, {:request, cat, hngr, c1, c2, self()})

  # thread-blocking
  receive do
    :denied ->
      meow(cat, krnlPID, c1, c2, food, base_hngr, hngr - 1,
    laziness)

    {:approved, c1, c2} ->
      :timer.sleep(1)

      # return chopsticks
      send(krnlPID, {:return, c1, c2})

      # start thinking
      :timer.sleep(laziness)

      # thinking over, back to eating
      meow(cat, krnlPID, c1, c2, food - 1,  base_hngr, base_hngr,
    laziness)
  end
end
```

How our cat brain works

Notable is that with this way of implementation, our algorithm is scalable. Meaning we can add more cats without changing the structure of our algorithm.

5 Benchmarking

Benchmarking such an application is quite tricky. The benchmark questions provided are, as I would argue, moot for this case since a lot of the issues with deadlocking are eliminated by the kernel sending a success/fail message allowing the cats to wait indefinitely.

The system, as mentioned before, is scalable. Since we have implemented a min-priority queue, we can even drop in a new cat on the fly.

For benchmark's sake, we add a `sleep` after every `:denied` request from the kernel. This would decrease our throughput throughout the system although make it more energy-efficient (if we even care about it). It certainly would not make it more memory since every `:denied` request is discarded from the queue.

To be more aggressive would mean to lower the hunger values for each cat. This would hardly affect the overall system time performance although it may cause some of the lower hunger valued cats to finish eating quicker with the risk of them dying in the process.

6 The decentralised solution

The decentralised solution involves not a kernel but rather every chopstick being it's own process.

The cats can send requests to the chopsticks on the side. We can allow our cats to send requests for them simultaneously.

In Elixir, functions with the same name (regardless of it's signature or parameters) share the same PID.

This means in our method where we wait for our left chopstick to send a response message, we do not need to fire another message to the left chopstick.

`send`: *pending* sends a message to the chopstick process and waits for a response (through the thread-blocking `receive` block). If it receives a message from the left chopstick first, it calls `send`: $left\_ok$ to now begin a wait for a response from the right chopstick.

If the first chopstick (whichever it may be) does not approve the request, we resend all our messages.

On some occasions, resending messages might cause the chopstick to send two response messages, the second message would appear as though the chopstick is unavailable. But we wouldn't have to care much about that because our cat would be way on it's journey of eating and sleeping.

```
# if we receive msg from left chopstick first
def send(:left_ok, c1, c2, c1Pid, c2Pid, b_hngr hngr) do
  receive do
    {c2, :approve} -> eat(); return(); sleep()
      send(:pending, c1, c2, c1Pid, c2Pid, b_hngr, hngr)
    {c2, :denied} ->
      send(:pending, c1, c2, c1Pid, c2Pid, b_hngr, hngr - 1)
  end
end
# if we receive msg from right chopstick first
def send(:right_ok, c1, c2, c1Pid, c2Pid, b_hngr, hngr) do
  receive do
    {c1, :approve} ->
      eat()
      return()
      sleep()
      send(:pending, c1, c2, c1Pid, c2Pid, b_hngr, hngr)
    {c1, :denied} ->
      send(:pending, c1, c2, c1Pid, c2Pid, b_hngr, hngr - 1)
  end
end
# if neither messages has been sent
def send(:pending, c1, c2, c1Pid, c2Pid, b_hngr, hngr) do
  send(c1Pid, {:request, self(), hngr})
  send(c2Pid, {:request, self(), hngr})
  receive do
    {c1, :approve} -> send{:left_ok, c1, c2, c1Pid, c2Pid, b_hngr,
    hngr}
    {c2, :approve} -> send{:right_ok, c1, c2, c1Pid, c2Pid, b_hngr,
     hngr}
    {cpstckAtm, :denied} -> send{:pending, c1, c2, c1Pid, c2Pid,
    b_hngr, hngr - 1}
  end
end
```

Requesting for chopsticks in a decentralised environment