Dining Philosophers

Oscar Melin

2018-03-24

1 Task

In this assignment several philosophers are sitting around a dinner table each with a bowl of food in front and one chopstick between each person. We thus have five philosophers, five chopsticks and one task to coordinate the use of our limited number of chopsticks.

A philosopher is either dreaming or eating. When a philosopher decides to eat, she picks up the chopsticks next to her (if they're not used by someone else) and starts eating. When done eating, she returns the chopsticks. The problem arises when multiple philosophers decides to eat at the same time.

1.1 Implementation

In order to simulate this problem, I've implemented a simple program following the general structure provided in the assignment instruction. The program is divided into three modules: A *chopstick* module (1.1), a *philosopher* module (1.1.2) and a main module called *dinner* (1.1.3).

1.1.1 A chopstick

A chopstick is represented by an erlang process that can be in one of two states; **available** and **gone**. When initialized, the chopstick will start in the state **available** while listening for a request message. If a request message is received the chopstick returns an 'ok' message and moves to state **gone**. While in this state, a chopstick can receive a 'return' message to return to state **available**. Regardless of state, a 'quit' message terminates the process. Figure 1 depicts the different chopstick states and transitions.

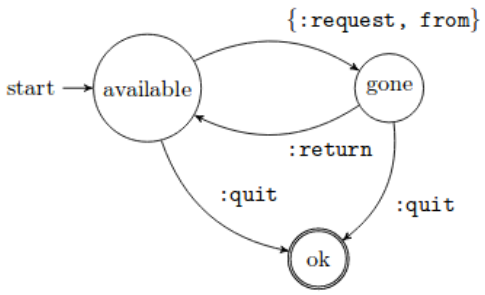

Figure 1: Finite state machine of the chopstick states. Source

1.1.2 A philosopher

A philosopher is similarly implemented as another erlang process that can be in one of two states; **dreaming** and **eating**. In the **dreaming** state the philosopher does nothing for a certain time until she decides to eat. If both adjacent chopsticks are available she will transition to the **eating** state. After eating both chopsticks are returned and the philosopher will return to the **dreaming** state. However if either of the adjacent chopsticks are unavailable, the philosopher continues to dream for a while until she wants to eat again. This process repeats until the philosophers 'Hungry' level reaches 0 after which she quits and leaves the table.

Time spent eating and dreaming is implemented using library functions timer.sleep/1 and function:rand.uniform/1 with a random seed that I calculate by utilizing the now deprecated built in function now/0.

1.1.3 Dinner time

The dinner module spins up all the chopsticks and philosophers as well as makes sure that each process terminates after execution.

```
start() ->
  spawn(fun() -> init() end).

init() ->
  C1 = chopstick:start(),
  C2 = chopstick:start(),
  C3 = chopstick:start(),
  C4 = chopstick:start(),
  C5 = chopstick:start(),
  Ctrl = self(),
  philosopher:start(2, C1, C2, "1 Arendt", Ctrl),
  philosopher:start(2, C2, C3, "2 Hypatia", Ctrl),
  philosopher:start(2, C3, C4, "3 Simone", Ctrl),
```

```
  philosopher:start(2, C4, C5, "4 Elizabeth", Ctrl),
  philosopher:start(2, C5, C1, "5 Ayn", Ctrl),
  io:format("~n------------~n"),
  wait(5, [C1, C2, C3, C4, C5]).

wait(0, Chopsticks) ->
  lists:foreach(fun(C) -> chopstick:terminate(C) end, Chopsticks),
  io:format("~nAll done!~n");
wait(N, Chopsticks) ->
  receive
    done ->
      wait(N-1, Chopsticks);
    abort ->
      exit(abort)
end.
```

2 Experiments

In my initial implementation the philosophers would never drop a single chopstick until they had finished eating and both eating and dreaming times were constant. The result of this was an immediate deadlock where all philosophers held a single chopstick while waiting for another to become available.

3 Break the deadlock

The first instance of all philosophers being able to finish their dinner was when I set the eating and dreaming times to different values but still constant. The result of this was that they all ate in the same order in a circle around the table.

```
...
1 Arendt received two chopsticks
1 Arendt done
2 Hypatia received two chopsticks
2 Hypatia done
3 Simone received two chopsticks
3 Simone done
4 Elizabeth received two chopsticks
4 Elizabeth done
5 Ayn received two chopsticks
```

```
5 Ayn done
...
```

In my last iteration of this part of the assignment I added randomness to the time each philosopher spent dreaming and eating as well as a requirement where both chopsticks had to be present for a philosopher to pick them up and start eating. As this problem represents a real world problem of processes sharing resources, this implementation seems more useful to experiment on.

```
...
1 Arendt is dreaming!
2 Hypatia is dreaming!
3 Simone is dreaming!
4 Elizabeth is dreaming!
5 Ayn is dreaming!
5 Ayn is eating!
3 Simone is eating!
5 Ayn done, 1 remaining
5 Ayn is dreaming!
3 Simone done, 1 remaining
4 Elizabeth is eating!
2 Hypatia is eating!
3 Simone is dreaming!
2 Hypatia done, 1 remaining
...
```

4 Asynchronous requests

Instead of sequentially requesting chopsticks, I could save time by adjusted my request function to asynchronously request both chopsticks first and the wait for the replies. To accomplish this I had to add a granted function that does the waiting for the philosopher.

```
request(Left, Right, Timeout) ->
  Left ! {request, self()},
  Right ! {request, self()},
  granted(Timeout).

granted(Timeout) ->
  receive
        ok ->
    receive
            ok ->
```

```
      ok
    end
  after
    Timeout ->
      no
    end.
```

If a philosopher gives up, how do we keep track of which chopsticks that was actually obtained? Is this a tricky problem or a non-problem?

Using my implementation I can't keep track of which chopstick that was available. If I wanted to I could add a unique chopstick reference to the 'ok' message generated by the built-in function make_ref/0.

5 A waiter

Previously mentioned solutions to the deadlock problem provided ways to **prevent** a deadlock by removing one or more of the for conditions that need to hold for a deadlock to occur; Mutual exclusion, Hold-and-wait, No preemption and Circular wait [1].

By implementing a waiter that controls how many philosophers that can eat at any given time we are attempting to **avoid** a deadlock form occurring instead of preventing it by removing its preconditions. This is known as scheduling problem and requires global knowledge of all processes that need to be scheduled.

I implemented a very simple waiter that took advantage of the fact that no two adjacent philosophers can eat at the same time. Thus, if a philosopher requests to eat, the waiter will only give its approval if neither of the requesting philosopher's neighbours are currently eating. Using this method I'm able to avoid a deadlock from ever occurring.

A more advanced solution could utilize each philosopher's dreaming and eating times and create a priority queue based on an effective scheduling discipline.

6 Discussion

This was a very interesting assignment both in terms of learning about how to solve synchronization issues regarding concurrent programming as well as the programming aspect of setting up and managing asynchronous processes. As I'm not doing this assignment in sync with my original course

plan I already possessed some knowledge when it came to concurrency and scheduling which proved very helpful. Nonetheless, while doing this assignment I have learned a lot about the deadlock problem; why it occurs and how to prevent or avoid it. The best solution I found was to be mindful and aware about it's existence, implement chopsticks (locks) and therefor prevent if from occurring in the first place.

References

[1] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, "Operating Systems: Three Easy Pieces", link