CO545 Assessment 4 – Producer-Consumer problem

This assessment is about implementing a solution to the *Producer-Consumer problem*. In this classic concurrency problem, there are two processes, the *producer* and the *consumer*, that both have access to a shared *buffer* (a queue) of *finite size*. The producer repeatedly generates data and adds it to the end of queue. The consumer repeatedly removes data from the front of the queue. However, the producer must wait if the queue is full, and dually the consumer must wait if the queue is empty.

If care is not taken, then a race condition can occur leading to a deadlock: the buffer is full so the producer waits, meanwhile the consumer empties the buffer and then goes into a waiting state; both producer and consumer are then locked indefinitely. To avoid this, synchronisation is needed between the consumer and the producer. This assessment guides you to a solution in Erlang, where the buffer is a process itself that co-ordinates the producer and the consumer.

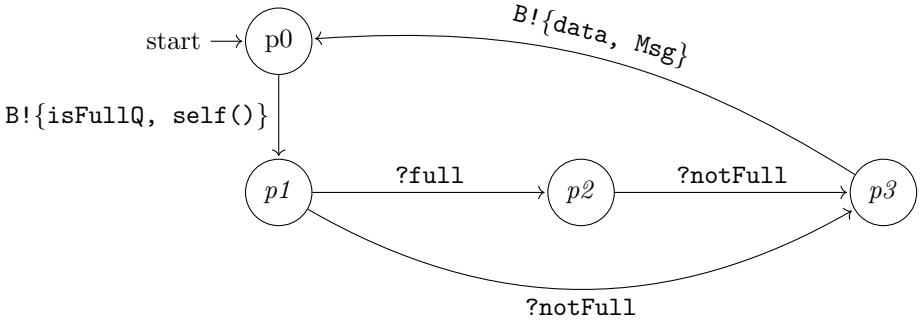
Download `producer.erl` and `tests.erl` from the Moodle. You must submit a single Erlang module named by your login ID, e.g., `dao7.erl`, which contains your solutions to the four tasks here, clearly marked with comments.

Your code should be valid Erlang (i.e., compilable) before submitting, otherwise it will be subject to a penalty.

To maximise marks, attempt every task, even if you have only a partial solutions to some.

Producer

A producer is provided in `producer.erl`, with behaviour described by the following CFSM:



Thus, the producer sends to a buffer `B` a request to see if it is full. If the buffer is not full, then the producer can send the buffer a new piece of data. If the buffer is full, then the producer must wait for a `notFull` message before it can proceed with sending its data. This is then repeated.

Task 1: Logger (6 marks)

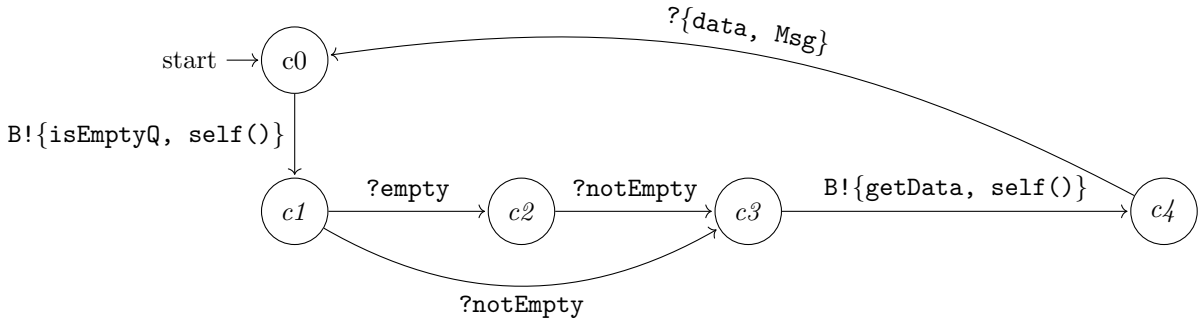
The producer also sends string message to a process `Logger` (not shown in the above CFSM).

Define a unary function `logger` for this process which repeatedly receives string messages and prints them out (e.g., via `io:fwrite`), counting the number of messages it has received and labelling its printed messages with this count.

For example, if the fifth message received is `"P: Buffer full I wait."` then the logger should print out `[5]:  P: Buffer full I wait.`

Task 2 : Consumer (15 marks)

Implement a consumer process described by the CFSM below, whose communication behaviour has a very similar shape to that of the producer:



The consumer sends to a buffer `B` a request to see if it is empty. If the buffer is not empty, then the consumer requests some data (`getData`) from the buffer and then waits to receive this data (transition $c4 \rightarrow c0$). If the buffer instead at state $c1$ the buffer is empty, then the consumer waits to receive notification that the buffer is not empty (transition $c2 \rightarrow c3$), before proceeding.

Include code to send messages to the logger process after every receive that explains what has happened so far (you can see a sample of appropriate messages shown in Task 4 below). Include a counter of the number of `data` messages received so far which is included in the message sent to the logger after receiving `data`, e.g. `C: Got data:  #5 = "Hello"`.

Hint: it may help to define this process using two functions: one function which covers transitions from state $c0$ to $c3$ and another function which covers the transitions from state $c3$ to $c0$.

Task 3 : Buffer (25 marks)

Implement the buffer such that it can handle all the messages sent by the above producer and consumer as described above. As a starting point you may use the following stub:

```
buffer(MaxSize) ->
  buffer([], MaxSize, none, none).

buffer(BufferData, MaxSize, WaitingConsumer, WaitingProducer) ->
  stub. % Delete me and write your code
```

where `buffer/4` has parameters:

- `BufferData` which is a *list* representing the queue stored in the buffer;
- `MaxSize` which is the maximum size of the queue (passed in from `buffer/1`);
- `WaitingCustomer` which is either `none` (meaning no customer is waiting) or it is the process ID of the waiting customer;
- `WaitingProducer` which is either `none` (meaning no producer is waiting) or it is the process ID of the waiting producer.

You need not send any logging messages from inside the buffer.

Test suite You should test your implementation against the small test suite provided in `tests.erl`. To run the tests from the Erlang shell do:

```
c(producer).
c(yourloginId).
c(tests).
tests:tests(yourloginId).
```

If all the tests pass you will see the output:

```
Test for simple interactions done.
Test for more complex interactions of producer and consumers done.
```

Otherwise you will see one or more messages describing which tests have failed. You can then find the source of these messages in `tests.erl` to understand what has gone wrong.

You will receive up to 10 marks for passing the tests. For every test that fails, this amount of marks will be decremented by 1, down to 0.

Note, the tests are indicative of the kind of interactions that should support. They are not exhaustive, i.e., you may still have problems even if all the tests pass. The rest of the marks for this task will be assigned based on the correctness of your code and its quality.

Task 4 : Main function (4 marks)

Finally, implement a nullary function `main` which spawns and connects the producer (parameterised so that it produces 5 data messages), your consumer, your logger, and your buffer (parameterised to have a maximum queue size of 5).

Running `main` should emit a log akin to the following modulo ordering and randomised data:

```
[0] C: Buffer empty. I wait.
[1] P: Putting data: 8
[2] C: Consumer awoke, asking for data.
[3] P: Putting data: 2
[4] C: Got data: #0 = 8
[5] P: Putting data: 3
[6] C: Asking for data.
[7] P: Putting data: 4
[8] C: Got data: #1 = 2
[9] P: Putting data: 9
[10] C: Asking for data.
[11] C: Got data: #2 = 3
[12] C: Asking for data.
[13] C: Got data: #3 = 4
[14] C: Asking for data.
[15] C: Got data: #4 = 9
[16] C: Buffer empty. I wait.
```