UNIVERSITY OF OXFORD
SOFTWARE ENGINEERING PROGRAMME

Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Tel +44(0)1865 283525 Fax +44(0)1865 283531
info@softeng.ox.ac.uk www.softeng.ox.ac.uk

Part-time postgraduate study in software engineering

Concurrent Programming, CPR

27th – 31st October 2014

ASSIGNMENT

The purpose of this assignment is to test the extent to which you have achieved the learning objectives of the course. As such, your answer must be substantially your own original work. Where material has been quoted, reproduced, or co-authored, you should take care to identify the extent of that material, and the source or co-author.

Your answers to the questions on this assignment should be submitted to:

Software Engineering Programme
Department of Computer Science
Wolfson Building
Parks Road
Oxford OX1 3QD

Alternatively, you may submit using the Software Engineering Programme website — `www.softeng.ox.ac.uk` — following the submission guidelines. The deadline for submission is 12 noon on Tuesday, 16th December 2014. If you have not already returned a signed assignment acceptance form, you must do so before the deadline, or your work may not be considered. The results and comments will be available after the next examiners' meeting, during the week commencing Wednesday, 29th April 2015.

ANY QUERIES OR REQUESTS FOR CLARIFICATION REGARDING THIS ASSIGNMENT SHOULD, IN THE FIRST INSTANCE, BE DIRECTED TO THE PROGRAMME OFFICE WITHIN THE NEXT TWO WEEKS.

---

1 Introduction

Through this assignment, you will demonstrate your understanding of concurrency, scalability and fault tolerance in Erlang-based soft real-time systems. You will do so by implementing the control center for a parcel delivery service.

The assignment is divided into a practical and a theoretical section. The practical section is a coding exercise for a soft real-time application. The theoretical section consists of questions on the application that you implemented, as well as other concurrency related items covered during the lectures.

2 Deliverables

While the practical deliverables are related, hand in a separate version of the source code for each exercise demonstrating the intermediate steps before achieving the final result. Do not submit your code in word or pdf format! Include an architectural description with the practical assignment, outlining assumptions you have made. And make sure you are strict in respecting the provided APIs. The practical assignment accounts for 70% of the grade.

In the theoretical section there are questions on the application that you implemented. When answering these questions, **please ensure you stay within the word limits specified**. The theoretical section accounts for 30% of the grade.

3 The Exercise

Your program is a nationwide parcel delivery simulator used to optimise delivery times and reduce transportation costs. It allows the user to generate data used to validate theories on where to place cargo stations and make decisions on the number of deployed vans and trucks.

Input data to the simulator consists of

- A list of cities with their population sizes (in thousands).
- A list of distances between pairs of cities. All the cities have to be connected, but not necessarily through a direct route – when going from one city to another, you might have to stop in other cities along the way.

When starting your simulation, you need to edit the configuration file and specify the simulation specific data. It consists of

- The cargo stations placed in selected cities.
- The number of trucks with a capacity of 20000 kg that transport the parcels between cargo stations and cities and the cities they are initially placed in.

- The number of courier vans with a capacity of 1000 kg that transport parcels between cargo stations and cities and the cities they are initially placed in.

The simulator will consist of a dispatcher informing the vans and trucks where to pick up and deliver parcels, as well as a journey planner, telling them which routes to take in-between cities. When the simulator is up and running, the system is ready to take in new parcel delivery orders. By testing different dispatch strategies, routes between cities and vehicle behaviour will provide metrics which allows us to optimise the service in relation to delivery times, capacity and fuel consumption.

We will break the exercise up into smaller parts that are easier to develop and test, combining them in a supervision tree.

3.1 Practical Part 1: Implementing a Journey Planner

Given a file containing a list of cities and distances between them, together with the simulation data, write a server which allows vehicles to provide a starting point and a list of destinations. The result will be a list of cities which the van or truck will have to travel through en route to picking up and delivering parcels.

The following API should be respected when implementing the server:

```
planner:start_link()  -> {ok, Pid}.
```

Start a new process that is linked to the calling process.

```
planner:route(From, ToList) -> {ok, CityList} |
                               {error, invalid}.
```

Given the name of a starting city and a list of destinations (in no particular order), this function returns a list that describes the route (and order) the truck or van has to take in order to visit all of the cities in `ToList`. The algorithm does not have to be optimal, but at the same time, it should not take you from Bristol to Cardiff via Edinburgh unless you have a parcel to deliver in Edinburgh. Keep in mind that there might not be direct routes between two cities, forcing you to travel via other cities. If cities or a route between two points does not exist, return `{error, invalid}`.

Hint: Make sure your route planner does not become a bottleneck, executing as much of the code as possible in the client process. Use the `file:consult/1` for the file operations. Read the manual pages for the file module.

3.2 Practical Part 2: Implementing the Parcel Manager

The parcel manager keeps track of parcels and provides vans and trucks with locations that require pickups. It can access the journey planner's tables and retrieve information on cities and routes as required.

The following API should be respected while implementing the manager:

```
manager:start_link() -> {ok, Pid}.
```

Starts a new manager process.

```
manager:send(From, To, Kg) -> {ok, Ref}.
```

If a client needs a parcel delivered, they call this function, passing in a string denoting the cities where the parcel needs to be picked up and delivered respectively. `Kg` is a number (integer or float) denoting the weight of the parcel. The manager returns a unique identifier, and when the parcel is delivered, sends a message of the format `{delivered, OrderId}` to the process that originally called `send/3`.

Vans and trucks should call the following functions to pickup and drop parcels:

```
manager:deliver(Loc) -> {ok,LocList}| {error,instance}.
```

A van or truck, upon starting its rounds, can provide its location and be given a list of destinations that have parcels which need delivering or picking up. Locations are provided based on proximity and non-reserved parcels that have been waiting the longest to be picked up.

```
manager:reserve(From, To, Kg) -> {ok, RefList}.
```

Returns a list of parcels weighing a maximum of Kg kilos in total that have been reserved to be picked up in a Location for delivery to a particular destination. This function returns the oldest parcels first and ensures that no other van or truck can pick them up.

```
manager:reserve(From, Kg) -> {ok, RefList}.
```

Returns a list of parcels weighing a maximum of `Kg` kilos in total that have been reserved to be picked up at a Location by a particular van or truck. This function returns the oldest parcels first and ensures that no other van or truck can pick them up. It provides unreserved parcels to all destinations available in `From` which should be transported to a cargo station.

```
manager:pick(Ref) -> ok |
                     {error, not_reserved | instance}.
```

If a van or truck has reserved the parcel, upon reaching its location, it uses this function to pick it up.

```
manager:drop(Ref) -> ok |
                     {error, not_ picked | instance}.
```

This function is used to drop off the parcel once the van has reached its destination. The `{delivered, OrderId}` message is sent to the process which originally called the send/3 function.

```
 manager:transit(Ref, Loc) -> ok |
                        {error, not_picked | instance}.
```

If the vehicle has picked parcels en route, it should terminate its round in the closest cargo station dropping off all parcels using this function. It can also call this function if it comes across a cargo station en route to its final destination, dropping off packages whose destination is not in the planned route.

```
manager:cargo(Loc) -> {ok, Loc} | {error, instance}.
```

A van or truck, upon having completed its deliveries, can provide its location and be given the closest cargo station where it can drop off all parcels it picked up en route and has not delivered. En route to the cargo station, there is nothing stopping the truck or van from picking up or delivering other parcels.

```
manager:lookup(Ref) -> {error, instance} |
        {ok,{Ref,From,To,Kg,Loc|VehiclePid,OwnerPid}}.
```

This function returns useful parcel information where a parcel is either in a particular location or in a vehicle. OwnerPid is the Pid of the client which will receive the `{delivered, OrderId}` message.

Keep your solutions simple, not optimal. You want an infrastructure that can be changed and improved as simulation metrics are being collected. The only important feature you need to implement is to ensure that parcels are not forgotten in any particular location and never picked up, and that when they are picked up, they do not travel in circles and actually reach their destination. The longer the delivery time, the higher the priority to pick up the parcel and deliver it to its recipient.

Test your solution with simple routes and locations.

3.3 Practical Part 2: Implementing the Vehicles

Create a process that can act as a van or a truck, the distinction between the two being the load each vehicle can take. Each van or truck process has to remember its load (the total weight cannot exceed its capacity), the reference and status of the parcels it is carrying, as well as their destination. It also needs to keep track of the current city. To move in-between connected cities, it has to wait for an amount of time proportional to the distance between the cities. Use the distance multiplied by 1ms for each leg of the journey.

To deliver a parcel, the following steps need to be taken:

1. Call `manager:deliver/1` to get a route assigned.
2. Reserve space for a set of parcel being picked up from one city to another, ensuring no other truck picks them up using `manger:reserve/3`.
3. The selected vehicle needs to appear in the sender's city and load the reserved parcels using `manager:load/1`.
4. When in a city, you can decide if you want to pick up other parcels within the vehicle's weight allowance using `manger:reserve/2` that have not been reserved or do nothing.
5. Decide if you want to transport the parcel to the receiver or a cargo station.
6. If the parcel is in the cargo station, decide if it should be transported to another cargo station closer to its final destination.
7. It is enough for a van to reach a city for a parcel to be picked up or delivered. Use `manager:drop/1` to deliver the parcel.
8. End your round in a cargo station dropping off all remaining parcels you picked up en route using `manager:transit/2`. Find your closest cargo station using `manager:cargo/1`.

A vehicle may deliver parcels and collect orders simultaneously. It can also wait for more parcels for any amount of time. It is up to you to choose a delivery strategy. It does not need to be optimal, but the delivery times and system throughput should be reasonable.

You can start with a simpler configuration file where there is only one station, a few cities and a couple of vans, expanding the scope of the simulation when you have verified that your strategy is working.

3.4 Practical Part 3: Implementing Fault Tolerance

We use supervision trees to implement our fault-tolerant strategy. To get a suitable supervision structure, we need to have different supervisors at different levels capable of handling static and dynamic children.

3.4.1 Vehicle supervisor

This supervisor manages vehicles. It should be in the module '`vehichle_sup`' and implement the following interface:

```
vehichle_sup:start_link() -> {ok, Pid).
```

Starts a new empty vehicle supervisor that is linked to the calling process.

```
vehichle_sup:stop(SupPid) -> ok.
```

Stop the phone supervisor and terminate all the linked vehicles.

```
vehichle_sup:add_vehichle(SupPid,{Loc,Kg}) -> {ok, Id}.
```

Start a new vehicle process and adds it to the supervision tree.

```
vehichle_sup:remove_vehichle(SupPid, Id) -> ok.
```

Remove the vehicle from the supervision tree.

If a vehicle process terminates abnormally, it should be restarted in the closest cargo station (where mechanics fixed it). Decide and document what happens with the parcels. They may not get lost, but could be unloaded in the closest city where the crash occurred, allowing other trucks to pick them up. Or you could be towed with the vehicle, which once repaired, continues its round. What is important is that a crash does not leave the system in an inconsistent state.

3.4.2 Top level supervisor

The top level supervisor, implemented in the `top_sup module` starts and monitors the vehicle supervisor, the journey planner and the parcel manager. Choose a supervision strategy that ensures the system is not left in an inconsistent state and allows the other components to recover.

Randomly terminate processes, ensuring that the system recovers. Describe your recovery strategy, documenting decisions that lead you to your final design.

Stress test your system to ensure it behaves in a predictable way under heavy load. Describe how you stress tested it and what load it is able to withstand.

3.5 Theoretical Part (30%):

Answer questions 1, 2 and 3 with a maximum of 100 words per question. Answer question 4 with a maximum of 200 words. Feel free to use diagrams to support your answers, if needed. Short concise answers are preferred to long answers.

1. When dealing with concurrency in distributed systems, what are the advantages of having asynchronous message passing?

2. What is the advantage of a selective receive in your concurrency model when implementing finite state machines?
3. How would you stress test your system in order to find bottlenecks? How would you detect these bottle necks?
4. Describe concurrency-related race conditions that can occur in your system, explaining how you have (or have not) addressed them.

4 Guidance

In the practical section, your solution should only use libraries from the Erlang/OTP distribution. Make sure you follow the interface descriptions! Demonstrate your code with the given test cases. The test cases act as a basic benchmark for requirement satisfaction, but aren't representative of all the things that can happen or go wrong in a live production system, especially borderline cases.

You may be able to find partial solutions to the problems on the web. I recommend that you don't use them, or at least, don't rely on them: they are often of dubious quality, they are likely to implement slightly different specifications, they typically won't help your critical review, and they won't help you learn about functional programming. Whatever you do, do make sure you make clear the source and the extent of any derivative material.

Finally, please structure your answer so that there is a cover sheet that contains *only* your name, the subject and date, and a note of the total number of pages. Do not put any answer material on the cover sheet; begin your answer on a fresh page. Avoid putting your name on any page except the cover page. Please number the pages and sections. Include a soft version of your code with your assignment report.

5 Assessment criteria

The practical part of the assignment consists of 70% of the final grade and is intended to evaluate:

- your ability to implement highly concurrent and distributed real-time systems in Erlang;
- your ability to express algorithms using appropriate data structures and elegant function definitions;
- your ability to write clear and simple code that is both scalable and fault tolerant.

The assessment of the theoretical question will count towards 30% of the final grade. You will be graded based on the clarity and conciseness of your answers. If you do not keep within the word limit, points will be deducted.