Assignment 2 – Concurrent programming in Go

In this assessment you will work on a problem, using synchronous and asynchronous message passing in Go. You will also be asked to reflect on the properties of your solutions. The work should consist of a zip file `xxx.zip` (where xxx is your id) and containing the following files:

- `xxx_part1.go` (your solution to part 1)
- `xxx_part2.go` (your solution to part 2)
- `xxx_part3.go` (your solution to part 3)

Please only include files of the solutions you are attempting. The zip file should be submitted on Moodle within the given deadline. Please read carefully the section "Important Notes" at the end of this document.

The dentist problem (part 1) - 50%

You need to implement a dentist studio system, run by one single dentist. The dentist is sitting in a treatment room, waiting to meet patients. Out of the treatment room there is a small waiting room for ***n*** patients to sit in a FIFO queue.

The dentist. The dentist checks for patients in the waiting room.

- If there are no patients, the dentist falls asleep.
- If there are is at least one patient, the dentist calls the first one in. The remaining patients keep waiting. During the treatment, the dentist is active while the patient is sleeping[1]. When the dentist finishes the treatment, the patient is woken up, and the dentist checks for patients in the waiting room. And so on …

The patient. The patient, upon arrival, checks if the dentist is busy with other patients or sleeping.

- If the dentist is sleeping, the patient wakes the dentist up and falls asleep while being treated. The patient is woken up when the treatment is completed, and leaves (i.e., terminates).
- If the dentist is busy with another patient, the arriving patient goes in the waiting room and waits (i.e., sleeps). When the patient is woken up, the treatment starts: the patient falls asleep until being woken up at the end of the treatment.

[1] In this assignment: "actively doing something for an amount of time" needs to be implemented as a time-consuming action (i.e., a sleep!) while "sleeping" needs to be implemented as a blocking send/receive on a communication channel.

- **Your task.** Model the dentist problem. You will need to create a dentist function, and a patient function. Use (synchronous or asynchronous) channels to synchronize the activities of dentist and patient.

Here are the signatures of the functions:

```
func dentist(wait <-chan chan int, dent <-chan chan int)

func patient(wait chan<- chan int, dent chan<- chan int, id int)
```

where *wait* is a channel of size *n*, and *dent* is a synchronous channel. For now, assume that there are at most *m* patients (i.e., they all fit in the queue).

Constraints/hints.

1. Use message passing (no shared memory, counters, arrays, …).
2. Use asynchronous channel *wait* of size *n* for the waiting queue of patients.
3. If patients arrive and *wait* is full (e.g., *m>n*) they will block (see Go's implementation of write operation on full channels) until a space is free. This is ok. For the moment, just let this be.
4. Use synchronous channel *dent* to model the state (sleeping/awake) of the dentist. The dentist should fall asleep while reading on *dent*, not while reading on *wait*.
5. Use synchronous channels, one for each patient, to model the state (sleeping/awake) of that patient.
    a. Function *patient* creates this as a fresh channel (type *chan int*).
    b. This fresh channel will be added to *wait* (type *chan chan int*) if the patient needs to stay in the waiting room.
    c. The patient sleeps while waiting to read on this fresh channel: (1) when waiting in the queue, and (2) when having the treatment done.
6. Allow the treatment to take some random time (using *time.Sleep()*). This needs to be at the dentist side (as the patient is sleeping while having a treatment done).

Testing your solution (the given main function).

```
func main() {
 dent := make(chan chan int)    // creates a synchronous channel
 wait := make(chan chan int, n) // creates an asynchronous
                                // channel of size n
 go dentist(wait,dent)
 time.Sleep(3 * time.Second)
 for i:=0 ; i<m ; i++ {
    go patient(wait, dent, i)
    time.Sleep(1 * time.Second)
```

```
 }
 time.Sleep(1000 * time.Millisecond)
}
```

To test the solution, run it several times, changing the size of *wait* and the number of *patients*. You can initially keep *n=m.* Then you should try with m>n, by creating the patients before the dentist, by having random delays while creating the patients, etc.

You should get something like the following for n=m=5:

```
Dentist is sleeping
Patient 0 wants a treatment
Patient 0 is having a treatment
Patient 1 wants a treatment
Patient 1 is waiting
Patient 2 wants a treatment
Patient 2 is waiting
Patient 3 wants a treatment
Patient 3 is waiting
Patient 4 wants a treatment
Patient 0 has shiny teeth!
Patient 4 is waiting
Patient 1 is having a treatment
Patient 1 has shiny teeth!
Patient 2 is having a treatment
Patient 2 has shiny teeth!
Patient 3 is having a treatment
Patient 3 has shiny teeth!
Patient 4 is having a treatment
Dentist is sleeping
Patient 4 has shiny teeth!

Program exited.
```

Submit your solution as one single file: `xxx_part1.go` (where xxx is your id)

Introducing priorities (part 2) – 25%

This part consists of two problems: a coding exercise (2.a) and a question (2.b). Please submit (2.b) as a code comment in the file part2.go.

(2.a) Some patients need emergency procedure and need to be prioritised. The dentist establishes a priority-based queue system that gives some patients priority over others. Modify your solution to part 1 so that instead of channel *wait* there are two channels *hwait* and *lwait* for high- and low-priority patients, respectively.

Function patient is passed wither a low priority or a high priority queue. See below for a sample main/initialization function:

```
func main() {
    dent := make(chan chan int)
    hwait := make(chan chan int, 100)
    lwait := make(chan chan int, 5)
    go dentist(hwait, lwait, dent)
    high := 10
    low :=  3

    for i := low; i < high; i++ {
        go patient(hwait, dent, i)
    }

    for i := 0; i < low; i++ {
        go patient(lwait, dent, i)
    }

    time.Sleep(50 * time.Second)
}
```

The dentist will check that there are no high-priority patients before serving low-priority patients.

A naive solution of the problem above may cause *starvation* of low-priority patients: in case of a constant incoming flow of high-priority patients the low-priority patients would never get served. We do not want that. A common technique to prevent starving in such situations is *ageing*.

Change the signature of the dentist so that: (1) it takes as input two waiting queues hwait and lwait, and (2) can **both read and write** on hwait (so that can handle the ageing).

```
func dentist(hwait chan chan int, lwait <-chan chan int,
dent <-chan chan int)
```

The dentist needs to set a timer e.g.,

```
timer := time.NewTimer(m * time.Millisecond)
```

and move a patient from *lwait* to *hwait* whenever m milliseconds have passed since the last read from *lwait*. You can observe timer deadlines as if they were interactions:

```
<- timer.C
```

Function `patient` should not change.

(2.b) Assume Go does not have a fair semantics. Can you identify one possibility of **starvation** in the scenario described in the problem statement of part 1? Your answer should be justified (you can refer to your own code if you wish). If you identify one, say what you could change your solution to make the system starvation-free. Your fix needs to be general with respect to the set-up scenario created by the main function (i.e., it cannot require assumptions on the relationship between the size of queues or number of patients initialized by main).

Submit your solution as one single file: `xxx_part2.go` (where xxx is your id)

The assistant (part 3) – 25%

This part consists of two problems: a coding exercise (3.a) and a question (3.b). Please submit (3.b) as a code comment in the file part3.go.

(3.a) The dentist is overwhelmed by the overhead of handling two queues (*lwait* and *hwait*) and decides to hire an assistant.

Modify your solution to part 2 by introducing a new goroutine "*assistant*" that communicates with the patients using the queues *hwait* and *lwait*, and communicates with the dentist using one single queue wait. The dentist will not see or act on the queues *hwait* and *hwait* but only receive patients on wait. You need to modify the function dentist and main accordingly. Here is a sketch of the signatures:

```
assistant(hwait chan chan int, lwait <-chan chan int, wait chan<-
chan int, … )

func dentist(wait chan chan int, dent <-chan chan int, … )
```

Feel free to make addition where you see the "…", but do not change the use of the waiting queues.

The patient functions need to remain as in part 2 (patients will wake up the dentist on channel dent, not the assistant).

(3.b) Can you identify a possibility of **deadlock** that affects part 2 but not part 3? Justify your answer.

Submit your solution as one single file: `xxx_part3.go` (where xxx is your id)

Check-up list

A good solution to parts 1, 2, and 3 will:

- work well, yielding something similar to given output sample, where provided and have the timeouts requested in the problem description.
- have the structure specified in the problem description (functions, signatures, etc.).
- be based on channel passing, i.e., use only the channels mentioned in the problem description for parts 1 and 2. For part 3 you can be more creative.

In addition, a good solution to part 2 and 3 will answer the give questions as a code comment. Your answers should be motivated with concrete arguments.

Important Notes

Late or Non-Submission of Coursework
The penalty for late or non-submission of coursework is normally that a mark of zero is awarded for the missing piece of work and the final mark for the module is calculated accordingly.

Plagiarism and Duplication of Material
Senate has agreed the following definition of plagiarism:

"Plagiarism is the act of repeating the ideas or discoveries of another as one's own. To copy sentences, phrases or even striking expressions without acknowledgement in a manner that may deceive the reader as to the source is plagiarism; to paraphrase in a manner that may deceive the reader is likewise plagiarism. Where such copying or close paraphrase has occurred the mere mention of the source in a bibliography will not be deemed sufficient acknowledgement; in each such instance it must be referred specifically to its source. Verbatim quotations must be directly acknowledged either in inverted commas or by indenting."

The work you submit must be your own, except where its original author is clearly referenced. We reserve the right to run checks on all submitted work in an effort to identify possible plagiarism, and take disciplinary action against anyone found to have committed plagiarism. When you use other peoples' material, you must clearly indicate the source of the material using the Harvard style (see http://www.kent.ac.uk/uelt/ai/styleguides.html). In addition, substantial amounts of verbatim or near verbatim cut-and-paste from web-based sources, course material and other resources will not be considered as evidence of your own understanding of the topics being examined. The School publishes an on-line Plagiarism and Collaboration Frequently Asked Questions (FAQ) which is available at:
http://www.cs.ukc.ac.uk/teaching/student/assessment/plagiarism.local

Work may be submitted to Turnitin for the identification of possible plagiarism. You can find out more about Turnitin at the following page:
http://www.kent.ac.uk/uelt/ai/students/usingturnitinsts.html#whatisTurnitin