Exercise in subject:	**DAT320 Operating Systems**
Title:	**Lab 4: Data Race Detection & Profiling**
Deadline:	**Sunday Sep 28 2014 23:00**
Expected effort:	**10-15 hours**
Grading:	**Pass/Fail**
Submission:	**Individually**
Remarks:	Links in this document are clickable.

University of Stavanger
Faculty of Science and Technology

1 Introduction

This lab exercise is divided into two parts and deals with two separate programming tools. The first part will focus on high-level synchronization techniques and will give an introduction to Go's built-in data race detector. You will use two different techniques to ensure synchronization to a shared data structure. The second part of the lab deals with CPU and memory profiling. We will analyze different implementations of a simple data structure.

2 Part 1: High-level Synchronization & Data Race Detection

In the first part of this lab we will focus on high-level synchronization techniques using the Go programming language. We will return to these topics also in lab 5, with low-level protection using inline assembly.

The Go language provides a built-in race detector that we will use to identify data races and verify implementations. A data race occurs when two threads (or goroutines) access a variable concurrently and at least one of these accesses is a write.

We will work on a stack data structure that will be accessed concurrently from several goroutines. The stack stores values of type `interface{}`, meaning *any* value type. The stack interface is shown in Listing 1 and is found in the file `common.go`. The interface contains three methods. `Len()` returns the current number of items on the stack, `Pop() interface{}` pops an item of the stack (`nil` if empty), while `Push(value interface{})` pushes an item onto the stack.

Listing 1: Stack interface

```
type Stack interface {
  Len() int
  Push(value interface{})
  Pop() interface{}
}
```

For this lab we will use the tests defined in `stacks_test.go` to verify the different stack implementations. The tests can be run using the `go test` command. We will run one test at a time. Running only a specific test can be achieved by supplying the `-run` flag together with a regular expression indicating the test names. For example, to run only the `TestUnsafeStack` function, use `go test -run TestUnsafeStack`. There are two type of tests defined for each stack implementation we will be working on. One test verifies a stack's operations, while the other is meant to test concurrent access using a race detector. Study the test file for details.

Tasks and Questions:

1. In the context of concurrent programming; what is a *critical section*?
2. Take a look at the `UnsafeStack` type and its associated methods in `stack.go`. Explain why this stack implementation cannot be safely shared between multiple goroutines.
3. The `testConcurrentStackAccess(stack Stack)` function is used to test concurrent access to every stack implementation we will be working on. Give a *short and concise* description of what this test function does.
4. As stated in the introduction, Go includes a built-in data race detector. Read `http://golang.org/doc/articles/race_detector.html` for an introduction and usage examples.
5. What command line flag is needed to enable the race detector when using the `go test` tool?
6. Run the `TestUnsafeStack` test with the data race detector enabled. Look at **one** of the warnings and inspect the stack traces. What are the two conflicting operations for your current run of the test?
7. The file `stack_sync.go` is a copy of `stack.go`, but the type is renamed to `SafeStack`. Modify this file so that access to the `SafeStack` type is synchronized (can be accessed safely from concurrently running goroutines). You can use the `Mutex` type from the `sync` package to achieve this.
8. Verify your implementation by running the `TestSafeStack` test with the data race detector enabled. The test should not produce any data race warnings.
9. Go has a built-in high-level API for concurrent programming based on Communicating Sequential Processes (CSP). This API promotes synchronization through sending and receiving data via thread-safe channels (as opposed to traditional locking).

   The file `stack_csp.go` contains a `CspStack` type that implements the stack interface in Listing 1 (but the actual method implementations are empty). The type also has a constructor function needed for this task. Modify this file so that access to the `CspStack` type is synchronized. The synchronization should be achieved by using Go's CSP features (channels and goroutines).[1] This will require some self-study if you are not familiar with Go's CSP-based concurrent programming capabilities. A place to start can be the introduction found at `http://golang.org/doc/effective_go.html#concurrency`. For inspiration on how to solve this specific task, look at Chapter 7.2.3 in *M. Summerfield, Programming in Go: Creating Applications for the 21st Century (Developer's Library), 1st ed. Addison-Wesley Professional, 5 2012.* The sub chapter is available as a PDF on It's Learning. Note that you should also ensure that the stack operations are implemented correctly. You can verify them by running the `TestOpsCspStack` test.
10. Verify your implementation by running the `TestCspStack` test with the data race detector enabled. The test should not produce any data race warnings.
11. **Optional exercise:** Implement a `ForcePop()` method for the `SafeStack` type **or** the `CspStack`. The method should pop an item off the stack, but if the stack is empty, it should block until an item is pushed onto it. For the `SafeStack` type an option may be to use the `Cond` type in the `sync` package. To use channels would be a natural choice if you choose to adjust the CSP-version.

[1] There is in this case an amount of overhead when using channels to achieve synchronization compared to locking. The main point for this task is to give an introduction on how to use channels (CSP) for synchronization.

12. **Recommended reading:**

    - The Go Memory Model: `http://golang.org/ref/mem`
    - Introducing the Go Race Detector: `http://blog.golang.org/race-detector`

2.1 Deliverable for Part 1

- Report: Answers to questions 1-3, 5 and 6 in the `ANSWERS.md` file.
- `stack_sync.go` and `stack_csp.go` with your modifications. Your implementations will be evaluated against the tests defined in `stacks_test.go` with the race detector enabled.

3 Part 2: CPU and Memory Profiling

In this part of the lab we will use a technique called *profiling* to dynamically analyze a program. Profiling can among other things be used to measure an application's CPU utilization and memory usage. Being able to profile applications is very helpful for doing optimizations and is an important part of Systems Programming. This lab will give a very short introduction to how profiling data can be analyzed. You may in future lab exercises be required to use profiling to improve and optimize programs.

Profiling for Go can be enabled through the `runtime/pprof` package or by using the testing package's profiling support. Profiles can be analyzed and visualized using the `go tool pprof` program.

We will in Part 2 continue to use the stack implementations from Part 1. The file `stacks_test.go` contains one benchmark for the three different implementations. Each of them uses the same core stack benchmark defined in the `benchStackOperations(stack Stack)` function. The stack implementations are not accessed concurrently so that the benchmarks can be kept reasonably deterministic.

Tasks and Questions:

13. The file `stack_slice.go` contains a stack implementation, `SliceStack`, backed by a slice (dynamic array). You will need adjust this implementation in the exact same way you did for the `SafeStack` type at Task 7 in Part 1. This has to be done to make the benchmark between the three implementations fair and comparable.

14. Read *Profiling Go Programs* at `http://blog.golang.org/profiling-go-programs`. This blog post present a good introduction to Go's profiling abilities. You should also look at and `http://golang.org/pkg/testing/` and `http://golang.org/cmd/go/#Description_of_testing_flags` for information on how to run the benchmarks and details about how Go's testing tool easily enables profiling when benchmarking.

15. Run the three stack benchmarks described earlier. Enable memory allocation statistics by supplying the `-benchmem` flag. In addition also use `-memprofilerate=1`. What is the purpose of this flag? Attach the benchmark output in your report. You can avoid running any tests together with the benchmarks by providing a non-matching regular expression to the `run` flag (for example `-run none`).

16. Which of the stack implementations uses the lowest average amount of time for the core stack benchmark? Which implementation allocates the least amount of memory and why? How does the CSP-based stack implementation compare in terms of CPU and memory usage against the other two?

17. Give a short description of the different *time-space* trade-offs involved when choosing to implement a stack backed by a linked list or a (dynamic) array.

18. Run the `BenchmarkCspStack` separately. This time you should also write a CPU profile to file when running it. Load the data in the `pprof` program. Attach the top ten and top ten cumulative listing of function samples to your report. Look at the top ten cumulative listing. What type of runtime functions dominates the listing below the basic `CspStack` methods?

19. Run the `BenchmarkSafeStack` separately and write a memory profile to file. Using `pprof` identify the only function allocating memory. List this function and identify the line number where the allocations occur. Also attach the output of this listing to your report.

20. **Optional exercise:** Explore the visualization possibilities offered by `go tool pprof` when analyzing profiling data.

3.1 Deliverable for Part 2

- Report: Answers to questions 16-19 in the `ANSWERS.md` file.
- `stack_slice.go` with your modifications.