Assignment 7: Lock-free approaches

ETH Zurich

1 Concurrent Logger

1.1 Background

Your company is developing an application server, which, for security reasons, has to log every action. The log is stored in memory and crawled by various intrusion detection algorithms. Since the current logger uses a lock—which is obviously a performance bottleneck—you decided to rewrite it using lock-free techniques.

A log entry contains a date and a message. The date, which also contains the time, is represented as a long integer. The message is a string. Adding a log entry should not require traversing the whole log.

Java provides several classes that support atomic compare-and-set, in particular, *AtomicLong* to store long integers, and *AtomicReference*<*V*> to store references such as String. These classes have three important methods:

```
// Atomically sets the value to the given updated value if the current value == the
    expected value. Returns true if successful.
boolean compareAndSet(V expect, V update);

//Gets the current value.
V get();

//Sets to the given value.
void set(V newValue);
```

1.2 Task

Your task is to implement this logger according to the interface below, without using locks:

```
public interface Logger {
  //Adds a new log entry
  void addLogEntry(long date, String message);
}
```

Hint: The data structure for the log should not have a fixed capacity and it should be possible to add retrieving and pruning methods later. You may define as many classes as you need for completing the task.

1.3 Solution

```
public class LockFreeLogger implements Logger {
    private final LogEntry first = new LogEntry(0, "Initializing logger");
    private volatile LogEntry almostLast = first;
```

```
    @Override
    public void addLogEntry(long date, String message) {
        LogEntry current = null;
        LogEntry newEntry = null;
        do {
            for (current = almostLast; current.next.get() != null; current = current.next.
                get());
            newEntry = new LogEntry(date, message);
        } while (!current.next.compareAndSet(null, newEntry));
        almostLast = newEntry;
    }
}

class LogEntry {
    public final long date;
    public final String message;
    public final AtomicReference<LogEntry> next = new AtomicReference<>();

    public LogEntry(long date, String message) {
        this.date = date;
        this.message = message;
    }
}
```

2 Spin Lock

2.1 Background

A spinlock[1] is a simple (but not very efficient) lock-free algorithm in which a thread trying to acquire a lock is made to wait in a loop while checking if the lock is free. It is an example of a busy-waiting algorithm because the thread waiting in the loop is not idle, but not doing any useful work either.

2.2 Task

You task is to write fill in the pseudocode for the *acquire* and *release* methods of the spinlock class given below. Assume that there is a function $compare\_and\_swap(target, old, new)$.

```
class SPINLOCK

  feature locked: INTEGER

  feature  make
    do
      -- write your code here.
    end

  feature acquire
    do
      -- write your code here.
     end

  feature release
```

```
16      do
          -- write your code here.
18      end
    end
```

2.3 Solution

```
1
    class SPINLOCK
3       feature
            locked: INTEGER
5
        feature make
7           do
                locked := 0
9           end

11      feature acquire
            local
13              stop: BOOLEAN
            do
15              from
                    stop := False
17              until
                    stop
19              loop
                    stop := compare_and_swap (locked, 0, 1)
21              end
            end
23
        feature release
25          do
                locked := 0
27          end
    end
```

3 Atomic update of multiple values

3.1 Background

An online game with thousands of players features a daily high score. The high score consists of the player's name and the score he or she achieved. Profiling determined that the current lock-based implementation is a bottleneck.

3.2 Task

You are asked to provide a prototype of a lock-free solution, pseudo-code is sufficient. You can use an integer for the score. Provide a routine to update the high score if the new score is better and a routine to retrieve the current high score. If you need additional data structures, describe them as well.

You may use atomic CAS: assume that there is a function $compare\_and\_swap(target, old, new)$.

3.3 Solution

```
-- A class providing the mechanisms for the daily high score
class HIGH_SCORE
    feature {NONE}
     -- The name and score of the player having achieved the highest score today. A tuple is
          used to be able to set it atomically.
        data: TUPLE[name: STRING, score: INTEGER]
    feature retrieve: TUPLE[STRING, INTEGER]
     -- Retrieve the name and score of the player currently havig the highest score.
        do
         --Atomic retrieval of the current high score. Creating a copy to ensure changes to
              the Result are not propagated.
            Result := data.copy
        end
    feature update (a_name: STRING; a_score: INTEGER)
     -- Checks the current high score and replaces it with the new score by the player
          named 'a_name' if 'a_score' is greater than the current high score.
        local
            l_data, l_new_data: like data
            l_success : BOOLEAN
        do
            from
                l_success := False
            until
                l_success
            loop
             -- Atomic retrieval of the current high score.
                l_data := data
                l_success := l_data.score >= a_score or else
                        compare_and_swap (data, l_data, [a_name, a_score])
            end
        end
end
```

References

[1] http://en.wikipedia.org/wiki/Spinlock