CS 636 Semester 2020-2021-2: Assignment 2

29th March 2021

Due Your assignment is due by Apr 12, 2021, 11:59 PM IST.

Policies

- You should do this assignment ALONE.
- Do not plagiarize or turn in solutions from other sources (including the Internet). You will be PENALIZED if caught.
- We MAY check your submissions with plagiarism checkers.

Submission

- Write your programs in C++ .
- Submission will be through mooKIT.
- Submit a compressed file with name "<roll-no>.tar.gz". The compressed file should have the following structure.

```
roll-no
--assign2-report.pdf
--<problem1-dir>
----files
--<problem2-dir>
----files
```

- The "`assign2-report.pdf`" file should contain explanations and results that have been asked for (if any) in the following problems. Explain the results to help the evaluators.
- The assignment requires you to fork an existing Git repository. After forking, invite all the TAs to the forked Git repository to help with the evaluation.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.
- You are encouraged to use the LaTeX typesetting system to generate the PDF report.

Evaluation

- Write your code such that all the strategies and the EXACT output format are respected.
- We will evaluate your implementations on a Unix-like system, for example, a recent Debian-based distribution.
- We will evaluate the implementations with our OWN inputs and test cases, so remember to test thoroughly.

Problem 1 **[50 marks]**

You are required to develop a high-performance airline booking system.

There is a single producer thread that enqueues SEATS seats (defined in the template file `airlinebooking.h`). The thread enqueues the seats one by one in ascending order (starting from 1) and exits.

At the same time, CUSTOMERS consumer threads (defined in the template file `airlinebooking.h`) are trying to get vacation tickets. Each consumer thread demands `n` seats at a time where `n` = (`rand()` % 4+1), i.e., each consumer can demand at most 4 seats. A consumer thread attempts to dequeue the `n` seats one by one. A consumer is done if `n` seats are available and allocated to the consumer. If 0<`k`<`n` seats are available, the consumer thread will keep retrying until there are no more seats left.

After a consumer thread has successfully booked `n` seats, it should print "Consumer with ID I has booked `n` seats with seat numbers S_1, S_2, ...,S_n", where $1 \leq S_i \leq$ SEATS is a seat number. In case a consumer needs `n` seats but the plane is full after having booked `n-k` ($k > 0$) seats, then the thread should print "Consumer with ID I has booked `n-k` seats out of `n` with seat numbers S_1, S_2.. S_{n-k}."

- The producer thread does not need to know what is happening to the consumer threads.
- The number of available seats can be a synchronized variable (say `std::atomic`).
- Use a global lock to execute the print statements in a mutually exclusive manner.
- Provide a `dump_queue()` method to print the state of the queue. The method should disallow concurrent modifications to the queue when it is invoked, to ease with debugging and evaluation.

The main thread will join on all the consumer and the producer threads. Once all the child threads are done, the main thread (i.e., the program) should exit.

You are supposed to implement and use a **lock-free** MS queue. You should read the popular MS queue paper and implement the algorithms presented.

Problem 2

[100 marks]

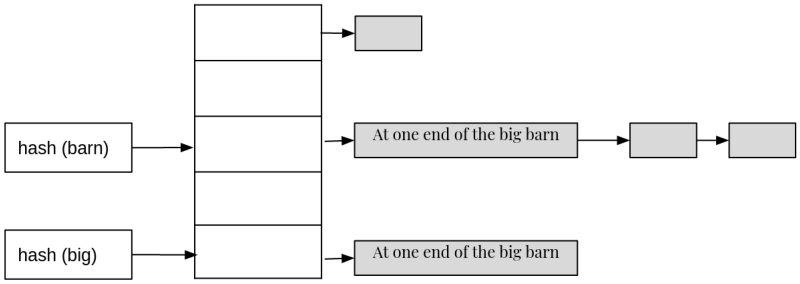
You are required to implement concurrent hash table designs to facilitate lookup of all sentences containing a given word.

The input to your program will be a file containing a list of comma-separated triples. The first member is the operation `[I-insert, R-remove, F-find]`, the second is the word, and the third is a sentence containing the word in case of an insert operation. A sample input file format is given below.

Input format.

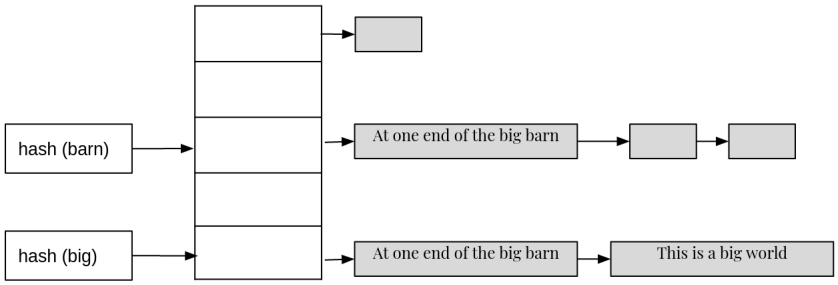
```
I,one,At one end of the big barn
I,end,At one end of the big barn
R,one
I,big,At one end of the big barn
F,one
I,barn,At one end of the big barn
```

Your hash table should use `hash(word)` as key into the hash table and `sentence` as value as shown in figure . The key will be of `MAX_KEY` and value will be of `MAX_VALUE` size as provided in the template code. The `sentence` should be inserted at `hash(word)`, refer to the figure . Your implementation should use chaining to resolve key collisions. The `hash()` function to generate key is provided in the template code.

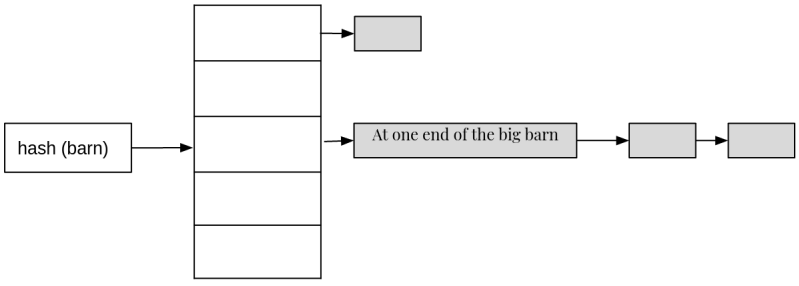


You are expected to provide following APIs for the hash table.

- **insert(word, sentence)** – Inserts the sentence at `hash(word)` locations in the table. `insert(big,This is a big world)`



- **find(word)** – Returns all the sentences containing the word, one sentence per line. `find(big)` returns
  At one end of the big barn
  This is a big world
- **remove(word)** – Removes the word and associated sentence from the hash table. `remove(big)`



Part I (40 Marks) Implement a blocking concurrent hash table data structure. You should use locking at the finest granularity such that operations to the same hash table location should

only compete for acquiring a lock. Your implementation should not lock the entire hash table for operations, doing so will result in deduction of marks.

Size of the hash table is provided as `SIZE` in the template code. You don't have to handle hash table resize as part of this assignment.

`find(word)` should block if an `insert` or `remove` operation is going on for that word in the hash table.

Part II (60 Marks) Implement a non-blocking version of concurrent hash table in Part I. You should NOT use locks in this part of the implementation. Note that lock free implementations are often vulnerable to the ABA problem. You can maintain metadata information to avoid the ABA problem.

Test scenarios.

- You program will be given input files equal to number of threads. The number of threads is defined by `THREADS` in template code. For example if `THREADS = 4`, then the program will take 4 input files, `input1.txt, input2.txt, input3.txt, input4.txt`, each input file containing data as per the format mentioned.
- We will test your implementation for different `THREADS` values
- Your implementation will be tested for correctness by checking the state of the hash-table after performing concurrent `insert()` and `remove()` operations.

Error conditions.

- If insert(word, sentence) fails, return `ENOSPACE`
- If find(word) fails, return `ENOWORD`
- If remove(word) fails, return `ENOWORD`