

CS 610 Semester 2024–2025-I: Assignment 5

4th November 2024

Due Your assignment is due by Nov 16, 2024, 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not copy or turn in solutions from other sources. You will be PENALIZED if caught.
- You can use any concurrency library to help with your implementation. The data structure implementation obviously has to be your own.

Submission

- Submission will be through Canvas.
- Submit a compressed file called “`<roll>-assign5.tar.gz`”. The compressed file should have the following structure.

```
-- roll
-- -- roll-assign5.pdf
-- -- <problem1-dir>
-- -- -- <source-files>
-- -- <problem2-dir>
-- -- -- <source-files>
```

- We encourage you to use the L^AT_EX typesetting system for generating the PDF file. You can use tools like Tikz, Inkscape, or Draw.io for drawing figures if required. You can alternatively upload a scanned copy of a handwritten solution, but MAKE SURE the submission is legible.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Evaluation

- Write your programs such that the EXACT output format (if any) is respected.
- We will primarily use the GPU3 department server for evaluation.
- We may evaluate the implementations with our OWN inputs and test cases, so remember to test thoroughly.

Problem 1

[80 marks]

Implement a concurrent open-addressing-based hash table using Pthreads. Implement support for the following operations.

Listing 1: Operations supported by the bounded partial stack data structure

```
1 void batch_insert(HashTable *ht, KeyValuePairs *kv_pairs, bool* result);
2 void batch_delete(HashTable *ht, KeyList *key_list, bool* result);
3 void batch_lookup(HashTable *ht, KeyList *key_list, uint32_t* result);
```

The third parameter `result` is a boolean array to indicate the success of an insert (duplicate keys are not allowed) or delete query. For `batch_lookup()`, `result` will contain the values if present, and a negative value otherwise to indicate a failure. Adapt the driver to the exact prototype that you design.

Description

- Key and the value are four bytes (unsigned int) each.
- The APIs work on a batch of items.
- Use double hashing for resolving collisions. You should test different hash functions and compare their performance.
- Assume that the concurrency is limited to operations in a batch. Each batch contains operations of the same type. For example, there can be concurrent insertions of 10000 key and value pairs in a batch.
- Use threading to issue concurrent calls to the hash table data structure (e.g., OpenMP).
- Assume the maximum load factor is 0.8.
- We will provide random keys and values (via files) that will be input to the hash table for testing.
 - `random_keys_insert.bin`
 - `random_values_insert.bin`
 - `random_keys_delete.bin`
 - `random_keys_search.bin`
- Empirically ensure the correctness of your code with unit tests. The unit test cases can contain a fixed sequence of operations. Include the test cases as separate function calls (e.g., `test1` and `test2`).
- TBB's implementation is open source and should be installed under `/usr/include/tbb`. Use version ≤ 2020 . CSEWS1 workstations have version 2021 installed. You can use newer versions if you can get it to work without any dependency on oneAPI.
- Use compile time flag `USE_TBB` to switch between the two versions.
- Use the best hash functions (determined empirically from bullet (iii)) for comparing with TBB.

Submission

- (i) A header file that implements the hash table,
- (ii) Extend the driver code to make calls to the hash table and compute the throughput of the kernels (e.g., 1000 inserts per second),
- (iii) Compare the throughput of your implementation with different primary and secondary hash functions (include all hash functions in the header file),
- (iv) Extend the driver to use a concurrent hash table using Intel TBB,
- (v) Implement a print method which can be invoked after every operation (e.g., `batch_insert()`) to check the content of the hash table,
- (vi) Compare the throughput of the TBB and Pthread implementations.

Problem 2

[70 marks]

Implement an “unbounded, total, lock-free” concurrent stack using linked lists. The stack will contain only positive 4-byte integer values.

Listing 2: Operations supported by the unbounded total lockfree stack data structure

```
1 int pop();
2 void push(int v);
```

Since the Stack is unbounded and total, we do not need APIs like `isEmpty()` and `isFull()`.

Listing 3: Possible definition of a linked list node

```
1 class Node {
2     public:
3     int value;
4     Node *next;
5 }
```

- Unlike Problem 1, concurrency means different threads can make individual calls to `push()` and `pop()`,
- Your implementation should be non-blocking, i.e., you should NOT use locks,
- You can maintain a pointer to the top of the stack as follows: `std::atomic<Node*> top`,
- A `pop()` operation on an empty stack should return a negative integer, i.e., it will not block,
- Ignore OOM issues while implementing the `push()` operation,
- Use `random_values_insert.bin` from Problem 1 to generate a sequence of random values to be pushed,
- Whether a thread issues a `push()` or a `pop()` can be decided based on probability,
- Lock-free implementations are vulnerable to the ABA problem. An easy fix that is mostly correct is to maintain the count of pop operations performed on the stack. A CAS operation should then compare both the `top` pointer and the count.

Submission

- (i) A header file that implements the concurrent stack,
- (ii) Extend the driver code from Problem 1 to work with stacks,
- (iii) Report the time taken to complete n concurrent operations, where $n \in \{1e5, 1e6, 1e7\}$,
- (iv) Implement a print method which can be invoked from the driver to check the content of the stack,
- (v) Evaluate the scalability (strong) of your implementation with threads (e.g., 1,2,4,8, and 16).