# Cangjie Concurrency Challenge

*Limits: 600 sec., 4096 MiB*

## Background

Cangjie is a new general-purpose programming language. We strongly recommend familiarising yourself and learning the language before starting the tasks.

The main user manual can be found here. We also have created some further tutorials:

- https://aliftrs.github.io/cangjietutorial.io/
- https://github.com/magnusmorton/openharmonyASPLOS
- https://github.com/Huawei-Edinburgh-Programming-Languages/TAsummary/blob/main/summary.md

### Setting up

Please visit https://cangjie-lang.cn/en/download and download the STS (0.53.18) version of Cangjie.

- MacOS and Linux.

  Extract the tarball then, in the terminal in the extracted directory, type

  `source envsetup.sh`. This needs to be done for each terminal session.

  You should now be good to go. Simply type `cjc <filename> -o out` to invoke the compiler. Type `./out` to run the executable.

- Windows.

  In addition to the above, the Windows package will have `envsetup.bat` and `envsetup.ps1` scripts that achieve the same as `envsetup.sh`.

## Problem

In modern high-concurrency systems, data is constantly being updated while multiple threads access it simultaneously. This creates strict requirements for correctness and consistency. Even a simple key-value store becomes challenging to implement in a multithreaded environment: we must ensure that all reads and writes appear as if they happened in some single, globally consistent order.

The goal of this task is to implement, in Cangjie, a concurrent in-memory key-value store (similar in spirit to Redis) that safely supports concurrent operations. You are required to provide an interface with `put` and `get` operations while guaranteeing **linearizability** — a strong consistency model where each operation appears to take effect instantaneously at some point between its invocation and its completion. This problem forms the foundation for building reliable distributed systems and is a key exercise in concurrent programming.

There are two subproblems in this problem.

## Sub-problem 1

### Description

The task is to build a concurrent in-memory key-value store in Cangjie. The key-value store should provide the following API:

```
package techarena

public class KeyValue {
    public init()
    public func put(k: String, v: Int64): Unit
    public func get(k: String): Option<Int64>
}
```

- `init()` returns a new instance of *KeyValue*.
- `put(k: String, v: Int64): Unit` adds a new key-value pair to the store or updates the value if the key already exists, where $k$ is the key and $v$ is the value.
- `get(k: String): Option<Int64>` retrieves the value associated with the given key $k$. This method should either return `Some(v)` if the key-value pair $(k, v)$ pair exists, or `None` otherwise.

### Correctness and scoring

The key-value store has to be linearizable. It means that even though operations may happen concurrently, each one appears to take effect instantaneously at some point between its start and end. From the point of view of all clients, the operations appear to occur in a single, consistent order that respects real-time ordering. You can read more about linearizability here: https://en.wikipedia.org/wiki/Linearizability.

You **must not** use any data structures from the *std.collection.concurrent* library, nor *std.collection.HashMap*, *std.collection.HashSet* or *std.collection.TreeSet*. Failure to comply will result in a **failed submission (Wrong Answer 1)**. All submissions may be **reviewed later**, and any detected violations may lead to **disqualification or annulment of results**.

Solutions will first go through a validation process to ensure that the `put` and `get` operations are linearizable. The implementation must not crash or deadlock when executed concurrently. Validation failure will result in a failed submission.

Scoring will be based on the number of operations that your key-value store can successfully execute in a set time period. The testing script spawns a fixed number of Cangjie threads that call `put` and `get` methods concurrently.

Important notes:

- You may assume all keys are no more than 100 characters in length.
- Each key string consists of digits and lowercase or uppercase English letters.
- There will be no more than $10^6$ different keys.
- You should not assume a uniform distribution of keys in `get` and `put` calls; some keys may be used more frequently than others.
- Some keys will contain other keys as prefixes.
- You are free to choose your own data structures and algorithms.
- The scoring script will be run on a 4 core machine.
- The scoring script will be run for a fixed *astronomical* time.

## Sub-problem 2

The second part is to implement serialization and deserialization for the data structure in part 1 by providing the following API:

```
package techarena

public class KeyValue {
    public init()
    public func put(k: String, v: Int64)
    public func get(k: String) : Option<Int64>
    public func getKeys(): Array<String>
    public func serialize(): String
    public static func deserialize(s: String): KeyValue
}
```

- `getKeys(): Array<String>` returns an array containing all the keys in the key-value store. Duplicates are not allowed in the array, and this array does not need to be sorted.
- `serialize(): String` converts the key-value store into a string.
- `deserialize(s: String): KeyValue` reconstructs a key-value store from the input string.

### Correctness and scoring

Correctness requires that calling `deserialize` on the output of `serialize` must yield a `KeyValue` instance with exactly the same keys and values as the original. The function `getKeys()` will be used to verify that both key-value store instances contain same set of keys. Failure to produce the exact same keys and values will result in a failed submission.

Note that these three functions, `getKeys()`, `serialize()` and `deserialize(String)` are **not** required to be thread-safe; they will only be executed in a single thread. However, you are encouraged (but not required) to attempt to optimize their implementation on multiple threads.

Scoring will be based on the total *astronomical* execution time of loading the key-value store with values, serialization, deserialization and extracting values for all keys and the length of the serialized string.

Important notes:

- You may assume all keys are no more than 100 characters in length.
- Each key string consists of digits and lowercase or uppercase English letters.
- There will be no more than $10^6$ different keys.
- Some keys will contain other keys as prefixes.
- You are free to choose your own data structures, algorithms and serialization format.
- The scoring script will be run on a 4 core machine.
- The scoring script will measure *astronomical* execution time of your solution.

## Final Scoring

Your solution can get one of the following verdicts:

- `Compilation Error` — your code didn't compile. You can see compiler error message on `Compiler` tab of your solution. Please make sure that you don't include `main` function in your code.
- `Time Limit Exceeded` — your solution deadlocked while handling concurrent requests, or took too long to serialize or deserialize.
- `Memory Limit Exceeded` — your solution used too much memory.

- `Runtime Error` — your solution crashed.
- `Wrong Answer` — your solution failed to satisfy the linearization requirement, or fails to produce the same set of keys or key-values after deserialization or we have detected the usage of the forbidden libraries.
- `Accepted` — your solution is correct. Well done!

You get 0 points for any verdict other than `Accepted`. If your solution is correct, the final score is calculated as follows:

$$score = \left\lfloor \frac{qps}{\sqrt{(t_s + t_d) \cdot len}} \cdot 10^4 \right\rfloor,$$

Where:

- $qps$ is the average number of concurrent `get` and `put` queries that your solution can process in one *astronomic* second,
- $t_s$ is the *astronomic* time in seconds that your solution needs to load the specific number of keys and serialize them into a string,
- $t_d$ is the *astronomic* time in seconds that your solution needs to deserialize the serialized string and extract all values for all keys,
- $len$ is the number of symbols in the serialization string.

## Final Notes

- The memory limit is 4096 Mib. Please note that the testing system will use up some small portion of this memory limit.
- The code size limit is 64 Kib.
- The compilation time limit is 1 minute.
- There is only one test case that combines the scores for two sub-problems.
- You can submit your code no more than once every 6 hours, and you will get feedback with your provisional score.
- While the provisional scores might be a bit volatile, the final scores after the final testing will be much more precise.
- The final results will be announced in one week.
- The testing will be performed on a machine with 4 cores.

## Quick start

Please check the quick start code snippet which contains the definition of the class and all the methods. It is important that your solution has the same exact package, class and methods names and parameters. Failure to comply with this template will most likely result in the `Compilation Error` verdict. You can review the compiler output on the `Compiler` tab of your submission.

Also check out the quick start solution which implements a very simple binary search tree data structure.