Department of Information and Computer Science
Utrecht University

INFOB3CC: Assignment 1
IBAN Calculator

Trevor L. McDonell
Ivo Gabe de Wolff
Tom Smeding

Deadline: Friday, 29 November 2024, 23:59

Change Log

2024-11-04: Initial release

Introduction

Since the beginning of 2014, old-fashioned bank account numbers have been abolished in The Netherlands and only so-called IBAN numbers are used. Unfortunately, a bank called Venster is having difficulty adapting their system to the new schema. The purpose of the assignment is to write a program that returns information on the bank account numbers which the organisation manages. Because the range of bank account numbers is very large, your program must use multiple threads at the same time.

The m-test

Bank account numbers contain a checksum system, so not every nine-digit number is a valid account number. To test whether an account number is valid we use a modified eleven's test, hereafter the m-test.

In the eleven's test, all figures are added together, weighted according to their position, and the total tested whether it is a multiple of 11. The last digit is multiplied by 1, the second-to-last digit multiplied by 2, and so on. An example with bank account number 27.48.56.190:

$$2 \times 9 + 7 \times 8 + 4 \times 7 + 8 \times 6 + 5 \times 5 + 6 \times 4 + 1 \times 3 + 9 \times 2 + 0 \times 1 = 220 = 20 \times 11$$

In the m-test, the resulting sum must be a multiple of m (so, not always 11). Note that a bank account number does not necessarily have 9 digits, it can be more or less. The key is that the k-th digit from the right is always multiplied by k.

Input

With this assignment you will write a Haskell program that does input and output via the console. The input consists of the following space separated arguments:

1. An integer b with $0 \leq b < 2^{30}$, the inclusive lower limit of the search range.
2. An integer e with $b < e \leq 2^{30}$, the exclusive upper limit of the search range.
3. An integer m with $1 \leq m \leq 256$, the modulus to be used in the m-test.
4. An integer p with $1 \leq p \leq 256$, the number of threads that your program must use to count.
5. A string `u` which determines the program mode to be started, either "count", "list", or "search". See below for more information on each of the program modes.
6. If the program runs in search mode, a string `h` that is the SHA1 hash being sought.

Locking

The program must be able to search with multiple threads at the same time. The entire search range must be distributed fairly over the different threads, without gaps or overlap, as described in the section on the different program modes.

If you have a critical section in your program you should shield it. For this assignment you have to implement two methods for sharing values between threads, depending on the program mode:

- Mutable references from the module `Data.IORef`.
- Synchronised mutable variables from the module `Control.Concurrent.MVar`.

For each program mode (see below), check carefully which you are supposed to use: we are trying to help you practice with implementing locks on top of various different abstractions.

Read the documentation carefully in each of these modules to make an informed choice about how you can use those for synchronisation. It is not permitted to use structures other than `MVar`s and `IORef`s for this assignment.

Program modes

Depending on the command line arguments, your program must be able to operate in different modes. These are:

Counting mode In this mode you must count the number of integers x with $b \leq x < e$ that satisfy the m-test. The threads must maintain a *shared* counter where the final answer is stored. The final output is a single number: the number of valid account numbers in the specified range.

The program must be able to search with multiple threads at the same time; use the provided function `forkThreads`. The number of threads to start can be found in the `Config`.

The entire search range must be distributed fairly over the different threads, without gaps or overlap. The work is distributed as fairly as possible: if the range is not evenly divisible between the number of threads, make sure that the difference in the number of account numbers each thread looks at is at most one.

Each thread must compute the number of valid numbers within its range. It must add its count to a shared `IORef`. To synchronise access to this `IORef`, you must write a CAS loop. For counting mode, you can only use `IORef`s and the associated functions on them: `newIORef`, `readIORef`, `writeIORef`; you are *not* allowed to use `atomicModifyIORef` and similar functions in this mode. Instead, the package `atomic-primops` has been added to the template, from which you are allowed to use the `readForCAS`, `casIORef` and `peekTicket` functions (and no others). See their documentation[1] for how to use them.

It is not allowed to use `MVar`s in this mode (except in the function `forkThreads` which was provided as part of the template).

Your code for counting mode should scale well to many threads. Therefore, each thread should write to the `IORef` only *once*: writing to it many times will create too much contention.

List mode List mode is similar to counting mode, but for every valid bank account number that you find, the program should output a single line containing the sequence number and account number, separated by a space. When a valid account number is found, it must be written immediately; you are not allowed to save everything to be displayed at the end.

Again, the program must be able to use multiple threads at the same time. Every thread should output its own found numbers. In list mode, you must use `MVar`(s) to get the next sequence number, and also synchronise access to the output console (i.e. you should prevent that two threads are writing to the console at the same time). You cannot use `IORef`s for list mode.

Note: In order for the test suite to pick up your output for this mode, you should not print the output to the console directly (using `putStrLn` or `print`), but instead with `hPutStrLn handle`, using the `handle` we have given you as an additional parameter to `list`. When running the program normally (as illustrated in the section *Examples* below), this handle is connected to the console, allowing you to see what you are outputting as usual. When testing (using `cabal run iban-test`, see below), this handle is connected

[1] `https://hackage.haskell.org/package/atomic-primops-0.8.4/docs/Data-Atomics.html`.
Because Haskell, like C#, is a managed (garbage collected) language, you can not use the hardware instruction for compare-and-swap directly. The functions from `atomic-primops` wrap it in a safe API; the documentation explains how this works.

to an internal buffer in the test framework so that we can see, and check, what you output.

If you want to write additional helper functions for list mode, you can of course pass on the `Handle` to those helper functions so that you can output things there as well.

Search mode A European Union command has come in to block the account of a troublesome political leader. To prevent espionage this type of request never directly specifies the account number; it only contains the SHA1 hash and the range in which the account number is to be found. The hash is displayed in hexadecimal format, for example as seen on `https://caligatio.github.io/jsSHA/`.

In search mode you are given the SHA1 hash and must find the corresponding account number in the given range and satisfying the m-test. As soon as the correct account number is determined, the program should terminate all threads as soon as possible. The `search` function returns a `Maybe Int`, which is `Nothing` if no account number was found that satisfies the inputs, and `Just` with the correct account number otherwise.

Checking if a number satisfies the m-test is relatively cheap, but generating the SHA1 hash is relatively expensive. Because of how the m-test works, not every chunk of a certain number of candidate integers will have equally many valid account numbers in it. Thus, when you divide work over threads, the number of hashes per thread will differ, and hence the amount of work per thread.[2] Therefore, you need to implement some load balancing between the threads.

You must implement a *concurrent queue* to distribute work between the threads: while one thread is enqueueing an item to the front of the queue, it must be possible for another thread to simultaneously dequeue an item from the end (if the queue was already non-empty). This queue should contain work items, which are smaller ranges within the total range of numbers to check. The queue initially contains the total search range, and when a thread claims work from the queue, it will split the range into a small part which it can do itself, and a remainder, to be divided in half and put back in the queue as two separate work items. When a thread claims work which is already sufficiently small, it does not have to put work back in the queue, as it can do all the work itself. You can experiment a bit to find a good value for the cut-off point.

When all work has finished (the last thread has finished the last job) or as soon as the correct bank account has been found (without computing the remaining jobs on the queue), you must stop all threads. You must come up with a good (and safe!) way to do this. To get you started here are some hints to think about. You might have a shared variable to count the number of pending tasks that you increment or decrement when you enqueue work or when finishing a dequeued work item; when this counter becomes zero, then the current thread just finished the last job. If you can determine there is no work left to do, you should stop all the other threads; if threads are waiting on the queue but there is no more work to do, you need to signal to them somehow that the work has finished. Remember that you need to stop *all* other threads.

[2] The perceptive reader will note that the amount will only vary slightly. We ignore this for this assignment, because we want you to implement a concurrent queue.

You can use any functions to work with `IORef`s or `MVar`s. To build the queue, you can make good use of the blocking functionality of `MVar`s: the *Threads 3* lecture explains how to build such a queue. The graders expect you to be using this queue; if you want to implement something different, please first consider whether it satisfies all the requirements given above.

General remarks

Here are a few remarks:

- Make sure your program compiles using `cabal build`. Please do not submit attempts which don't even compile.
- Use the template from the website. It includes useful functions like `forkThreads`, which will start the specified number of threads. To use this function, it may help to write top-level helper functions, which perform the work of a thread. You can give this function additional arguments like `Config → IORef Int → Int → IO ()`, partially apply it and then pass it to `forkThreads`.
- Because Haskell is lazy, make sure that the actual computation is performed before going into a critical section. You can use the function `evaluate :: a → IO ()` within a do-block in IO, which assures that the value is evaluated before continuing with the next IO operations.
- Include *useful* comments in your code. Do not paraphrase the code, but describe the structure of the program, special cases, preconditions, et cetera.
- Try to write readable and idiomatic code. Style influences the grade! For example, use indentation of 2 spaces. If you prefer an automatic formatter, you could use e.g. ormolu.[3]
- In addition to correct operation, you will be graded on aspects such as the size of your critical sections, a good division of the search range, and how thread termination is handled.
- Efficiency (speed) of your implementation influences the grade.
- Copying solutions—from other people, the internet, or elsewhere—is not allowed.
- Find the starting framework on the website:
  `https://ics-websites.science.uu.nl/docs/vakken/b3cc/assessment.html`
- Submission is via BlackBoard; the BlackBoard assignment will be published in due course. **Only submit `src/IBAN.hs`. Hence, only *change* `src/IBAN.hs`, otherwise the graders will not be able to compile your code.**

[3]`https://hackage.haskell.org/package/ormolu`

- See the setup instructions at:
  `https://ics-websites.science.uu.nl/docs/vakken/b3cc/haskell-setup-instructions.html`
- The starting template includes a small test suite to check your program. You can run the tests via the command `cabal run iban-test`. To run a subset of the tests you can use the `-p` flag: for example, to run only the list-mode tests use `cabal run iban-test -- -p list`. (Do not forget the '`--`'!)

  It may of course be helpful to write some additional tests yourself.

Examples

- Count numbers in a small range with 11-test:

  ```
  cabal run iban-calculator -- 274856170 274856190 11 4 count
  ```

  Then the output is:

  ```
  2
  ```

- List mode also outputs the account numbers in the order in which they are found:

  ```
  cabal run iban-calculator -- 274856170 274856190 11 4 list
  ```

  The order of the account numbers is therefore random, but the line numbers (sequence number) are ascending:

  ```
  1 274856182
  2 274856174
  ```

- The program must also work with an m other than 11:

  ```
  cabal run iban-calculator -- 374856170 374856250 12 4 list
  ```

  These numbers meet the 12-test:

  ```
  1 374856173
  2 374856238
  3 374856181
  4 374856246
  5 374856202
  6 374856210
  ```

- Be sure to test this search entry to know that you are using the SHA1 hashing function correctly:

  ```
  cabal run iban-calculator -- 274856170 274856190 11 4 search ↩
      c736ca9048d0967a27ec3833832f7ffb571ebd2f
  ```

  With output:

  ```
  274856182
  ```

- Your program should also terminate if no corresponding account number is found:

  ```
  cabal run iban-calculator -- 274856170 274856190 11 4 search 77↩
      ba9cd915c8e359d9733edcfe9c61e5aca92afb
  ```

  With output:

  ```
  not found
  ```

Submission

- This is an individual assignment.
- The deadline for this assignment is **Friday, 29 November 2024, 23:59**. To submit your assignment, submit `src/IBAN.hs` to BlackBoard.
- For this assignment you must only edit the file `src/IBAN.hs`. Ensure that your `src/IBAN.hs` compiles when put into the starting-template as-is.

Grading

1. (3 pt) Implement the program mode *count* using `IORef`s.
2. (3 pt) Implement the program mode *list* using `MVar`s.
3. (4 pt) Implement the program mode *search* using a concurrent queue.