Exercise 2

This exercise has total 15 points. Each point amounts to one percent of the total module mark. You are not allowed to use any external library in your code. We might ask you to explain your code.

All assessment deadlines in this module are strict. Late submissions will get a mark of 0. All submissions must work on the virtual machine as specified.

This exercise addresses the challenges in concurrent programming. It will use the Binary Search Tree (BST) data structure that you have implemented in Exercise1. Multiple concurrent threads will perform read/write operations on the BST data structure.

Before you start Ubuntu, please set the number of processors to 2 from VM VirtualBox Manager-->Settings-->System-->Processor.

If you are using UTM, right-click on your virtual machine edit->system->show advanced settings and set CPU cores to 2.

After this change, if you run the 'lscpu' command from a terminal, it should show the number of CPUs = 2.

Part 1:

Use your BST code (bst.c file) from Exercise1 or use the model answer code that are attached to this assignment page and add the following new functionalities. Please note that the header file has changed in this assignment, and you may need to adapt your code. You may add helper functions to the bst.c file.

Task 1.1 Node* balanceTree(Node* root);

This function creates a new balanced BST from an input tree of the form made in assignment 1. Note that the BST in Exercise1 is unbalanced and inverted, and the output should be balanced and no longer inverted. See the tips section on how to create a balanced BST in page 5.

Task 1.2 float avgSubtree(Node *N);

This function returns the average of all the nodes. For example, if you pass the root of a BST, the function will return the average of all nodes that are present in the subtree rooted at N. The algorithm for this function will be somewhat similar to printSubtree() as both functions perform complete tree traversal; printSubtree() prints the node-values whereas avgSubtree() accumulates the node values and then finds the average value of them.

Inside the main() function in test_bst.c file, averages of an unbalanced BST and its equivalent balanced BST are computed. If the implementation is correct, both averages will

be equal (note: the averages may be slightly different due to the implementation of the average as a float. Differences smaller than 0.001 will be fine.)

[Optional: Observe the clock cycle counts for the two average calculations in main(). In both cases, avgSubtree() visits the same number of nodes. So, in 'theory', both average calculations should roughly consume the same number of cycles. In 'practice', do you see any difference in the cycle counts? If you see a difference, what could be the reason? You do not have to submit your answers for this optional question. (run "make cycle" to do this experiment)]

Part 2:

Assume that a server machine manages a BST data structure. During the uptime, many clients perform operations on the BST. Each client sends its BST-commands in the form of a command- file to the server.

Valid commands that can be executed by the clients are:

(1) addNode <some unsigned int value>

The server will insert the node with the specified value in the tree. Note that this operation modifies the BST.

The server also prints a log using printf() in the format of

"[ClientName]insertNode <SomeNumber>\n"

(2) removeNode <some unsigned int value>

The server will delete the node with the specified value from the tree. Note that this operation modifies the BST.

The server also prints a log using printf() in the format of

"[ClientName]deleteNode <SomeNumber>\n"

(3) countNodes

The server will count the current number of nodes in the BST and print the number on the display. Note that this operation does not modify the BST.

The server also prints a log using printf() in the format of

"[ClientName]countNodes = <SomeNumber>\n"

(4) avgSubtree

The server will compute the average of all the nodes that are present in the BST and print the average on the display. Note that this operation does not modify the BST.

The server also prints a log using printf() in the format of "[ClientName]avgSubtree = <SomeNumber>\n"

The server executes the commands that are present in a command file one-by-one starting from the first line.

Example: Assume that the server has received a file 'client1_commands' from Client1. Let, the file contents be (starting from the beginning of the file):

```
addNode          19675   ...
...
avgSubtree
```

Thus, for Client1 the server will first insert a node with value 19675, then perform the following operations, and finally compute the average of the tree etc.

The server serves multiple clients concurrently. For each client, the server calls

`void* ServeClient(char *clientCommands)` in a concurrent thread and passes the name

of the command-file using the string pointer `clientCommands`.

Every 2 seconds the server has a downtime. During a downtime, the server transforms its BST into a balanced BST using the balanceTree() function. The reason for doing this during a downtime is to ensure that no nodes are added while setting up the tree, causing it to become unbalanced again. Note that different BST modifications during uptime causes imbalances in the tree. The main() function spawns a thread to perform the downtime operation every downtime.

The function prototype for executing downtime is as follows

```
void* downtime();
```

Task 2.1:

Implement the function in serve_client.c

```
void* ServeClient(char *clientCommands)
```

that reads the file specified by *clientCommands* for commands and executes the BST operations one-by-one according to the content of the file. Note that multiple clients will be executing BST operations concurrently. So care must be taken to avoid concurrency issues.

Task 2.2:

Implement the downtime() function in serve_client.c which calls the balanceTree() function every night. In this exercise, we assume that downtime happens every two seconds. So, balanceTree() is called every two seconds. You can use the function sleep(2) to cause a sleep of 2 seconds between two downtimes.

The program ends after three downtimes.

Code submission

Use the Makefile to compile your code. You are provided with a script called test.sh, that runs test_bst and compares the output with a reference output.

Use Valgrind to check memory leaks and errors. The Makefile produces test_bst.o file. The command for checking memory leaks will be valgrind --leak-check=full ./test_bst.o

Put bst.c and serve_client.c into a folder named with your student ID and **compress the folder** to a .zip file.

For example:

A student with student id of 1234567 should submit 1234567.zip and this file should be able to extracted to a folder with the following structure:

```
1234567/
├── bst.c
└── serve_client.c
```

Any other form of submission will not be accepted.

Marking scheme:

The program must not leak memory and not crash during execution. Any code which does not compile on the virtual machine provided will be awarded 0 marks and not be reviewed.

For marking we will use additional, more advanced, test scripts which check whether your program satisfies the specification. If the provided test scripts fail, all the more advanced test scripts are likely to fail as well

- 5 points are given for correctly implementing Task 1.1
- 1 point is given for correctly implementing Task 1.2
- 7 points are given for correctly implementing Task 2.1
- 2 points are given for correctly implementing Task 2.2

  We might ask you to explain your code.

Tips: How to create a balanced BST from an unbalanced BST?

There are several ways to balance a BST. The following approach is a simpler one. Perform inorder traversal along the given BST and store the node values in an array. Since inorder traversal produces a sorted sequence, the array will be sorted. Note that you have already implemented inverted inorder traversal in the printSubtree() function of Exercise1.

Next, build a balanced BST from the above created sorted array using the following recursive approach:

1) Get the middle element of the sorted array and make it the root of the tree.

2) Recursively do same for the left half and right half.

   a) Get the middle of the left half and make it the left child of the root created in step 1

   b) Get the middle of the right half and make it the right child of the root created in step 1.

You might need helper functions during the implementation.

Tips: How does the tests for this assignment work?

To test your code just run $ make test

Part1:

In test_bst.c , it takes an unbalanced tree and generates a balanced tree using the function you wrote. Then it computes the average of the nodes using the average function you provided, and the number of nodes using your previous number of nodes function. If these are different from one another or different from the expected value it will fail. If the test fails, the script will output the content of both trees on screen side by side.

Part2:

In test_bst.c, it will read files defined in *client_names*, then execute several clients in parallel. In the end, this should result in a tree that only has one node, the number in this node should be 1. The program checks if the tree is as expected in the end to determine if your code passes this primary test.

If your program doesn't handle concurrency very well, there is a chance that it may crash or get unexpected results in this test.