Exercise 2

This exercise has total 7 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.

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 and add the following new functionalities. 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 BST. Note that the BST in Exercise1 is unbalanced. See the tips section on how to create a balanced BST in page 5.

Task 1.2. int sumSubtree(Node *N);

This function returns the sum of all the nodes. For example, if you pass the root of a BST, the function will return the sum of all nodes that are present in the tree. The algorithm for this function will be somewhat similar to printSubtree() as both functions perform complete tree traversal; printSubtree() prints the node-values whereas sumSubtree() accumulates the node values.

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

[Optional: Observe the clock cycle counts for the two sum calculations in main(). In both cases, sumSubtree() visits the same number of nodes. So, in 'theory', both sum 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 day-time, 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) insertNode <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) deleteNode <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) sumSubtree

The server will compute the sum of all the nodes that are present in the BST and print the sum 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]sumSubtree = <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):

```
insertNode 19675
…
…
sumSubtree
```

Thus, for Client1 the server will first insert a node with value 19675, then perform the following operations, and finally compute the sum 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 night the server has a downtime. During a downtime, the server transforms its BST into a balanced BST using the balanceTree() function. Note that, different BST modifications during the day-time causes imbalances in the tree. The main() function spawns a thread to perform the downtime operation every night.

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 'night' happens every one second. So, balanceTree() is called every one second. You can use the function sleep(1) to cause a sleep of 1 second between two downtimes.

The program ends after three downtimes.

Code submission

Use the Makefile to compile your code. There is a script in 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 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

- 2.5 points are given for correctly implementing Task 1.1
- 0.5 points are given for correctly implementing Task 1.2
- 3.5 points are given for correctly implementing Task 2.1
- 0.5 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 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 sum of the nodes of each tree with the sum function you designed. 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. The balanced and unbalanced tree can also be found in b_tree.log and u_tree.log. The output generated by the program are stored in log_1

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 only have 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 pass this primary test.

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

You should also check log_1 to see if there are any concurrency problem.

Be aware, these are only primary tests, passing these tests doesn’t mean your code does not contain any bug. More extensive test cases will be used when grading this assignment.