ECEN 5033 Concurrent Programming

**FINAL PROJECT WRITE UP**

**Concurrent Tree**

# **Introduction:**

This document is about the concurrent binary search tree algorithm which has a key, value pair. It is used to insert nodes with keys and values, get values of nodes with mentioned keys, get keys of nodes i.e range between two node keys which are key\_begin and key\_last, and finally delete nodes from the tree in a concurrent manner. It takes values of from the different input files passed to the command line and produce output on the terminal, giving the keys, values in the tree, the range of nodes for the given output file and finally the print of all nodes in the tree.

**# Description of code organization:**

Consists of bst.c, Makefile, UnitTest.c

The final project code has been organized in three files which is the bst.c, Makefile and UnitTest.c. These two programs are different compiled with Makefile. It generates two executables bst and test on running make.

**# Description of every file**

* Makefile compiles the code bst.c, UnitTest.c and generates executable that can be run by typing make.
* bst and test contains the main two executables of the project.
* The bst program file takes the inputs as - - name, -p insertfile.txt, -g getfile.txt, -r range\_file.txt, -d deletefile.txt, - - lock=fg\_lock or - - lock=rw\_lock, -t as thread\_count.
* --name argument prints the full name of the person and lock argument takes options as fg\_lock or rw\_lock which concurrently performs insert, get, range and delete based on the the lock types selected.
* -t option takes input as number of threads which needs to be run in parallel .
* The locks used are hand-over-hand locks and read write lock.
* The Unit test program runs without any command line inputs.
* The bst.c contains all the binary tree functions for adding, searching for nodes. This file contains function used to read.. Reading the file is done by using char \*\*array and converting the array to an integer 1d array. The char \*\*array which reads input of from the unsorted file uses dynamic memory allocation and reallocates for file content greater than size of initially defined array.
* The read items which are key and value are used as inputs from file scanned using fcanf.
* In the program the elapsed time at the end will be printed on terminal in sec and nanosec.

# **Implementation Strategy for Binary search tree using hand over hand locking**

typedef struct node

{

pthread\_mutex\_t lock;

int key;

int value;

struct node \*left, \*right;

}Node;

* The Nodes of binary search tree consist of the right and left node, which could be either null or not null.
* All nodes in the tree consist of a key and a value which is added to them acting as index and values.
* All nodes are also initialized with mutex locks when they are created. These locks are later freed on end of program along with the entire tree.

typedef struct Tree

{

pthread\_mutex\_t lock;

Node \*root ;

}Tree;

* The Nodes consist of a structure tree which holds the main root of the binary search tree.

# **Implementation Strategy for Binary search tree using read write locking**

typedef struct Node\_rw

{

pthread\_rwlock\_t lock;

int key;

int value;

struct Node\_rw \*left;

struct Node\_rw \*right;

}Node\_rw;

* The Nodes of binary search tree consist of the right and left node, which could be either null or not null.
* All nodes in the tree consist of a key and a value which is added to them acting as index and values.
* All nodes are also initialized with read write locks when they are created. These locks are later freed on end of program along with the entire tree.

typedef struct Tree\_rw

{

pthread\_rwlock\_t lock;

Node\_rw \*root ;

}Tree\_rw;

* The Nodes consist of a structure tree which holds the main root of the binary search tree.

**# Common Strategy for Binary search tree using both locks**

* In both cases the tree can insert, get values of keys, get range of keys with the help of locking. When a new node is created, it is protected through locks of the upper node in the hierarchy. The root of the tree has its own lock which works appropriately in the case of no nodes in the tree and when first node is created. Using both these types of locking scheme, we prevent any data race even when multiple threads are operating.

typedef struct data\_thread\_iterations

{

int iteration\_number;

int iteration\_length;

int iteration\_item;

}data\_thread\_iterations;

* For both locks threads are spawned with following attributes assigned to them which are iteration number – thread id, iteration length – length of task, iteration item – task value.
* Insert method – This operation is used to insert new node to the tree with a unique key and value. The arguments required for this function is the Tree, which is initialized in main at the start along with its locks. The other arguments to this function are the respective key and values which are obtained from the input insert file. This function works by checking if root is NULL, if it is NULL, new root is created. It creates new node between locks of upper nodes, thereby creating isolation from different threads trying to insert nodes with keys and value.

**void insert(Tree \*tree, int key, int value)**

* Get method – This operation is used to get node’s value based on the unique key. The arguments required for this function is the Tree, which is initialized in main at the start along with its locks. The other arguments to this function is the respective key which are obtained from the input get file. This function works by checking if root is NULL, if it is NULL, it returns, otherwise it finds the location of the node with the key requested and returns that node. It creates isolation from different threads trying to get nodes with locks.

**Node \* get(Tree \*tree, int key)**

* Range method – This operation is used to get node between unique keys passed as key\_begin and key\_last to the function which are the arguments to the function along with the node root. The respective keys are obtained from the input range file. This function works by checking if root is NULL, if it is NULL, it returns, otherwise it finds the location of the node between the start and end.

**void nodes\_beween\_range(Node \*root, int key\_begin, int key\_last, int thread\_number)**

* Delete method – This operation is used to delete node from the tree. The arguments required for this function is the Tree. The other arguments to this function are the respective key value which can be obtained from the input delete file. This function deletes nodes and changes the pointer of upper node or changes the root depending on the node needed to be deleted.

**void deleteid(Tree \*tree, int key, int value)**

# **Parallelization Strategy**

* The number of threads will vary for different functions. Each thread performs operations such that they work on equal number of tasks, which is number of items divided by thread\_count – 4 for insert. The threads will have their unique iteration length-size of task, iteration number - thread id, item value – the task key.
* Then the respective insert, get, range handlers are invoked which run all of their operations concurrently.
* Divider for threads, that is number of tasks or length of task for which the thread will run depends on the size of file and thread count assigned to the functions.
* New threads are created, and their specific indexes and unique parameters are passed to the handler functions
* The threads are joined on completion of their unique tasks
* Finally time taken by all of them is printed to the screen.

**# Compilation instructions**

* Eg of how to run the binary search tree project. First compile the file using **make** which creates an executable of the name **bst**.
* Eg of how to run the Unit test. First compile the file using **make** which creates an executable of the name **test**.

**# Execution instructions**

* Then run the **bst** executable using the command by typing
* **./bst - -name -h -p insert.txt -g search.txt -r range.txt -d delete.txt -t 10 --lock=<fg\_lock, rw\_lock>** which concurrently creates, searches the tree with the nodes having particular key and values.
* Then run the **test** executable using the command by typing

**./test** which tries to verify working of all the operations insert, get, delete, range and prints if they succeeded.

**# Perf analysis Discussions**

* Analyzing high and low contention is done by inputting values in files in a such a way that in high contention only one node key is being accessed in the entire file, which means all threads will have to come to the same node to find the value for get and between same set of keys for range.
* On the other hand the data value for file testing low contention will be spread out for different indexes, that is different keys are used such that all threads will be accessing different nodes.

**Low contention files with hand over hand locks using 10 threads and 1000 line nodes with duplicates**

* Performance counter stats for './bst -p insert.txt -g search.txt -r range.txt -d delete.txt -t 10 --lock=fg\_lock':

279,803 L1-dcache-load-misses 4.22% of all L1-dcache hits

6,631,412 L1-dcache-loads 95.78%

5,641,949 branch-instructions 98.59%

79,696 branch-misses 1.41% of all branches

167 page-faults

0.028937850 seconds time elapsed

* Running Instruction for low contention hand over hand locks

./bst -p insert.txt -g search.txt -r range.txt -d delete.txt -t 10 --lock=fg\_lock

**Low contention files with reader writer locks using 10 threads and 1000 line nodes with duplicates**

* Performance counter stats for './bst -p insert.txt -g search.txt -r range.txt -d delete.txt -t 10 --lock=rw\_lock':

347,228 L1-dcache-load-misses 5.19% of all L1-dcache hits

6,696,438 L1-dcache-loads 94.81%

5,431,923 branch-instructions 98.61%

75,284 branch-misses 1.39% of all branches

200 page-faults

0.007711102 seconds time elapsed

* Running Instruction for low contention read write locks

./bst -p insert.txt -g search.txt -r range.txt -d delete.txt -t 10 --lock=rw\_lock

**High contention files with hand over hand locks using 10 threads and 1000 line nodes with duplicates**

* Performance counter stats for './bst -p insert.txt -g get\_high\_cont.txt -r range\_high\_cont.txt -d delete.txt -t 10 --lock=fg\_lock':

8,071,363 L1-dcache-load-misses 3.94% of all L1-dcache hits

204,779,860 L1-dcache-loads 96.06%

179,830,179 branch-instructions 99.25%

1,339,737 branch-misses 0.75% of all branches

1,035 page-faults

0.176894892 seconds time elapsed

* Running Instruction for high contention hand over hand locks

./bst -p insert.txt -g get\_high\_cont.txt -r range\_high\_cont.txt -d delete.txt -t 10

- - lock=fg\_lock'

**High contention files with reader writer locks using 10 threads and 1000 line nodes with duplicates**

* Performance counter stats for './bst -p insert.txt -g get\_high\_cont.txt -r range\_high\_cont.txt -d delete.txt -t 10 - - lock=rw\_lock':

862,102 L1-dcache-load-misses 4.31% of all L1-dcache hits

20,000,511 L1-dcache-loads 95.69%

12,594,561 branch-instructions 99.01%

125,224 branch-misses 0.99% of all branches

1,108 page-faults

0.011080891 seconds time elapsed

* Running Instruction for high contention hand over hand locks

./bst -p insert.txt -g get\_high\_cont.txt -r range\_high\_cont.txt -d delete.txt -t 10 - - lock=rw\_lock'

* It can be observed that for high and low contention nodes the page fault is little more in rw locks based binary search tree than hand over hand locks
* However, it is seen, rw\_locks have more L1 Data cache hits than hand over hand lock based binary search tree.