Design Document: Asg3

Jinghao Shen
CruzID: jshen30

1 Goals

The goal for Assignment 3 is to modify the multi-threaded RPC server to implement recursive name resolution, provide immediate persistence for key-value pairs, and improve overall performance and scalability. It will still provide all of the original functionality from earlier assignments, as well as the following additional features.

1. Ability for a variable to contain a variable name or a value. This includes setting and retrieving these names.
2. Recursive name resolution: if a variable contains a name, look up that name and use its value instead. Repeat until it hits a number, or until it exceeds a limit.
3. Variables will be persistent before the server acknowledges an operation.
4. Support for millions of variables (key-value pairs).
5. Improving performance of the server.

Original functions includes:

1. Math functions
2. Read from a file
3. Write to a file
4. Get file size
5. Create a file
6. Multi threads
7. Support for variables in math operations.
8. Sharing of the key-value store across all server threads, and synchronization between the server threads for reading and writing key-value pairs.

2 Design

Modified OLD Functions:

Making a hash table for items with a fixed-size char array as the name and a 64-bit signed integer or variable name as the value. The hash table has functions insert, replacement, delete, and lookup.

To store the key-value

```
struct HT_item{
        char* name;
        int64_t value;
        char* value2;
        uint8_t flag;
        HT_item* next;
}

struct hashTable{
        size_t size;
        HT_item** array;
```

```
        count;
    }

    import DJBHash
```

Create linked hash table items. It has “Next” to reference to the item linked to the current item.

```
    createItem(key, value)
        Name = key
        If (value == string)
            value2 = value
            value = NULL
            flag = 1
        if(value == number)
            value = value
            value2 = NULL
            flag = 0
        Next = null
        Return item
```

Create hash table:

```
    createTable(size)
        Table = array[size]
        return hashTable


    freeTable(hashTable*)
        for(0 to size)
            if(item != null)
                table[index]->null
```

Use hashFunction to get an index. Store the value to that index. If there already exists an item with the same variable name, update the value. If there already exists an item with the different variable name, link the current item to the end of the item link.

```
    Insert(hashTable*, key, value){
        newItem = createItem
        Index = hashFunction(key);
        if (array[index] == NULL){
            if(count > size)
                Table full return error
            insert newItem to array[index]
            count++;
        }else{
            if(key == array[index] -> key){
                update value;
            Else
```

```
                    curr = array[index]
                    while(curr -> next != null)
                        curr = curr-> next;
                    curr-> next = newitem;
```

Update the value if there is an existing variable name.

```
replacement(hashTable*, key, value)
    index = hashFunction(key);
    if(no such key)
        return
    else
        if(array[index]-> next == null && key == key)
            array[index]-> value = value
        else
            while(go through linked list){
                if(key == key){
                    update value
                }
            }
    }
    return
```

Go to the index on the hash table and go through each item if there's a linked list. If the variable name is founded, remove that item from the hash table.

```
delete(hashTable*, key)
    index = hashFunction(key);
    if(no such key)
        return
    else
        if(array[index]-> next == null && key == key)
            Array[index] = null
            count--;
        else
            if(array[index]-> key == key){
                remove array[index]
                set array[index]->next to array[index]
                count--;
                return

            while(go through linked list){
                if(key == key){
                    Remove from chain
                }
            }
```

```
        }
        return null
```

Go to the index on the hash table and go through each item if there's a linked list. If the variable name is founded, return the value of that variable. If nothing is found, return a magic number 1234567890.
Recursive function lookUpR is for recursive math function. It takes two more parameters, time to record how many times iterated and I as the maximum time. Return magic number if times out.

```
lookup(hashTable*, key)
        index = hashFunction(key);
        Item = array[index]
        while(item != Null){
                if(item->key = key)
                        return item->value
                if(item->next = null){
                        return 1234567890
                item = item->next
        return 1234567890

lookUpR(hashTable*, key,times,I)
        if(times >= I)
                Return 9876543210;
        Times += 1;
        index = hashFunction(key);
        Item = array[index]
        while(item != Null){
                if(item->key = key)
                        if(value = number)
                                Return value;
                        if(value = variable name)
                                Return lookUpR(hashtable,varName,times, I);
                if(item->next = null){
                        return 1234567890
                item = item->next
        return 1234567890
```

Go through all items on the index. If the index is not empty, write the variable name and value to the file. Create the file, if no destination is found.

```
dump(file)
        Open file
        for(i=0 to size){
                if(array[i] != null){
                        dprintf("varname=value\n")
```

```
                        if(array[i]->next != null)
                            while(go through linked list)
                                dprintf("varname=value\n")
                            }
                        }
                    }
                }
                close(file)
```

Open and read from the file. Get the variable and the value. Insert into the table.

```
load(file)
    Open file
    while(read)
        Extract variable name
        Extract value
        If variable name is malformed
            Return error
        insert(table, name, value)
```

To have multithread:

```
struct Thread {
    sem_t mutex;
    int cl;
};
```

Thread_data holds the thread and the shared hashtable

```
struct thread_data {
    Thread * t;
    hashTable* ht;
}
```

Initialize the list of threads and set them all to waiting. If a client sends a request, find the available thread in the list. Awake the thread.

```
Create threads[N] where N taken from argv
Create thread_data[N] which has thread and hashtable
Create mainMutex
    Initialize threads n times
    thread_data -> thread = threads[i]
    thread_data -> thread = ht (n times)
    pthread_create(...start...thread_data)
Start the connection
while(true){
```

```
            Cl = accept()
            while(available thread == 0){
                  wait(mainMutex);
            }
            threads[i].cl = cl;
            signal(mainMutex);
            wait(mainMutex);
      }
```

Retrieve thread info and hashtable from main. If awakened, do the process. After the process, return to waiting.

```
start(void* arg){
      td = arg
      while(true){
            wait(thread->mutex);
            process(cl, ht);
            cl = 0;
            signal(mainMutex);
      }
```

Process the command from client

```
process(cl, hashtable)
      read
      process recvBuffer
      write
```

New Features:

setv function. Get the name of variable and variable name to set to. Insert that to the hashtable.

```
      function(buffer[], operator, ifError){
            varLenX;
            varNameX;
            varLenY;
            varNameY;
            insert(varNameX, varNameY);
            return;
      }
```

getv function. Get the name of variable. Search in the hashtable. Return 2 if no such variable nama, 14 if the value is not a variable name, variable name if found.

```
      function(buffer[], operator, ifError){
            varLen;
            varName;
            varValue = lookUp(varNameX);
```

```
        if(varValue == int64_t)
            Return error(14);
        Else if(==not found variable name)
            Return error(2);
        Else
            Return varValue;
        return;
}
```

For Math functions, the input is separated into seven cases. Then, retrieve the value from the hashtable based on the variable name. If a 0x80 flag is encountered and the value of that variable name is another variable name, set the variable name to the value variable name and do a loopUp again until the value is a number. Calculate the result based on the input operator and return the result to the main function:

```
    function(buffer[], operator, ifError){
        if(del){
            Read 1 operand
            Remove that value name from hashtable
            return;
        }
        Switch
            Case 0x10
                A as operand
            Case 0x20
                B as operand
            Case 0x30
                A as operand
                B as operand
            Case 0x40
                res as operand
            Case 0x50
                A as operand
                res as operand
            Case 0x60
                B as operand
                res as operand
            Case 0x70
                A as operand
                B as operand
                res as operand
            Case 0x90
                A as operand
```

```
        while(A!=number)
                A=A.value
                loopUp(A)
Case 0xa0
        B as operand
        while(B!=number)
                B=B.value
                loopUp(B)
Case 0xb0
        A as operand
        B as operand
        while(A!=number)
                A=A.value
                loopUp(A)
        while(B!=number)
                B=B.value
                loopUp(B)
Case 0xc0
        res as operand
Case 0xd0
        A as operand
        res as operand
        while(A!=number)
                A=A.value
                loopUp(A)
Case 0xe0
        B as operand
        res as operand
        while(B!=number)
                B=B.value
                loopUp(B)
Case 0xf0
        A as operand
        B as operand
        res as operand
        while(A!=number)
                A=A.value
                loopUp(A)
        while(B!=number)
                B=B.value
                loopUp(B)
Default
        a = read 8 bytes from the buffer
        b = read another 8 bytes from the buffer
```

```
}


if(overflow)
        ifError = 22;
else
        ifError = 0;
if(there is a res as operand){
        insert(ht, key, value);
}
return a+b or a-b or a*b or a/b or a%b
```

To do Persistence and iterations check, first retrieve the -I and -d from the argument using getopt, the same way as -N and -H.

Send the value I to the process function and then to Math and lookUpR functions. The recursive function will end if the maximum iteration is reached.

To load the data from a directory, call loadDir with parameter hashtable and d after the hashtable is created. After each thread is completed, call dumpDir to store current hashtable to the directory d.