Hacettepe University

Computer Science and Engineering Department

Fall 2017

Name and Surname	***: Tolgahan DİKMEN***
ID Number	***: 21327929***
Course	***: BBM203 Programming Lab.***
Experiment	***: Assignment 2***
Subject	***: Stack, Queue, Dynamic Memory***
Due Date	***: 13.11.2017***
Advisor	***: R.A Burçak ASAL***
E-mail	***: tolgahandikmen95@gmail.com***

1. INTRODUCTION

In this assignment, a basic client-server architecture design is done using the C programming language. The design consists of two different parts. The first part starts creating a queue structure with dynamically which is given by user via .txt file, then the stack structure is formed in a similar way. The second part is to make the desired operations using these structures. The implementation has to be created dynamically because the sizes of the clients and the server are changeable according to the input file.

According to design, there must be only one server, but client number may change. Every client and server contains changeable size of queues and stacks. Each client operates on a specific set of processes or takes requests (interrupts) from a user. The client sends the requests and processes to a server and similarly, the server takes the processes sent by the clients and its own interrupts.

2. SOFTWARE USING DOCUMENTATION

2.1. Software Usage

The program needs two different inputs to work. The first input file contains size of the clients and sizes of the queues and the stacks for each client. The first line of the input file contains information on how many customers it is. The last line of the first input always belongs to server. The second input file contains run commands for the program.

There are four different operation commands in the program:

➢ **Process Request Command for the Clients:**
Format: ’A’ ’Client Number’ ’Process Character Name’
In this command, ’A’ character in the first slot means that there will be an addition to a specific client’s queue structure.
The second slot in this command specifies which client’s queue will a process be added to. It takes a number in the range of (1, Client size).

The third slot in this command specifies the name of the process that will be added.

- **Interrupt Request Command for Clients/Server:**

  Format: ’I’ ’Client/Server Number’ ’Interrupt Character Name’

  In this command, ’I’ character in the first slot means that there will be an addition to a specific client’s or the server’s stack structure.

  The second slot in this command type specifies which client’s (or whether the server’s) stack an interrupt will be added to. It takes a number in the range of (1, Client size+1).

  The third slot in this command specifies the name of the interrupt that will be added.

- **Send Command for the Clients:**

  Format: ’S’ ’Client Number’ ’G’

  In this command, ’S’ character in the first slot means that from a specific client, a process or an interrupt will be sent to the server, more specifically, it will be added to the server’s queue.

  The second slot in this command type specifies from which client, a process or an interrupt will be sent to the server. It takes a number in the range of (1, Client size)

  The third slot will always take ’G’ character, which means an invalid slot and it can simply be ignored.

- **Operate Command for Server:**

  Format: ’O’ ’G’ ’G’

  In this command, ’O’ character in the first slot means that the server will start to run its current process or interrupt.

  The second and third slot will always take a ’G’ character.

2.2. Error Messages

The program may return three type errors.

- **Error ‘1’**: If a specific client’s queue is full, the program gives an error character ’1’ when trying to process addition.
- **Error ‘2’**: If a specific client’s stack is full, the program gives an error character ’2’ when trying to interrupt addition.
- **Error ‘3’**: If a specific client’s stack is empty, but also its queue is empty as well, then the program gives an error character ’3’ when trying to sending/operating a command for the clients/server.

3. SOFTWARE DESIGN NOTES

3.1. Definition About The Values

Variables and arrays which has been used:

- **char *input = argv[1]:** Gets the name of the first input file which consist of sizes about clients and queues and stacks of the clients .
- **char *input2 = argv[2]:** Gets the name of the second input file which consist of operations about run commands.
- **char *output = argv[3]:** Gets the name of the output file path and name.
- **int stackSizesArray[ ]:** Holds stack sizes for each client and the server.
- **int queueSizesArray[ ]:** Holds queue sizes for each client and the server.
- **int client_size:** It equals to total clients size+1. ‘+1’ for server’s information.
- **int op_size:** It equals to operations size which has got from the input2.
- **int **operationArray:** It holds all operations in a two dimensional array.

- **Client *client[ ]:** It will be holds clients information.

3.2. Definition About The Functions and Structures

The program consists three structure of struct type.

```
typedef struct {

        char interruptName;

        int top;

} Stack;
```

That structure forms the basis of the stack structure. ‘top’ variable keeps track of changes. Always default ‘top’ variable is ‘0’. If any changes has been done, ‘top’ value is changed. ‘interruptName’ variable is holds to information about that stack’s interrupt.

```
typedef struct {

        char process;

        int front;

        int rear;

} Queue;
```

That structure forms the basis of the queue structure. ‘front’ variable keeps first element information of that queue and ‘rear’ variable keeps last element information of that queue. Always default ‘front’ and ‘rear ’variables are ‘-1’. ‘process’ variable is holds to information about that queue’s process.

```
typedef struct {

        Stack* stack;

        Queue* queue;

} Client;
```

That structure forms the basis of the client structure. Clients have one stack structure and one queue structure.

Functions about queue structure:

int isEmptyQueue (Queue* queue, int queueSize)

That function takes a queue and size of the queue as parameters. The function checks the given queue. If queue is empty, returns ‘1’, otherwise ‘0’

int isFullQueue (Queue* queue, int queueSize)

That function takes a queue and size of the queue as parameters. The function checks the given queue. If queue is full, returns ‘1’, otherwise ‘0’

int enQueue (Queue* queue, int queueSize, char process)

That function takes a queue, size of the queue and information of process as parameters. It adds process after to the last element (rear) of the queue. If operation is success, returns ‘1’, otherwise ‘0’

char deQueue (Queue* queue, int queueSize)

That function takes a queue and size of the queue as parameters. It deletes first element of the queue (from front) and returns process information of deleted element. If given queue is empty, the function returns ‘0’

Functions about stack structure:

int isEmpty (Stack* stack)

That function takes a stack as a parameter. The function checks the given stack. If stack is empty, returns ‘1’, otherwise ‘0’

int isFull (Stack* stack, int stackSize)

That function takes a stack and size of the stack as parameters. The function checks the given stack. If stack is full, returns ‘1’, otherwise ‘0’

int push (Stack* stack, int stackSize, char interruptName)

That function takes a stack, size of the stack and information of interrupt as parameters. It adds interrupt after to the last element of the stack. If operation is success, returns ‘1’, otherwise ‘0’

char pop (Stack* stack, int stackSize)

That function takes a stack and size of the stack as parameters. It deletes first element of the stack and returns interrupt information of deleted element. If given stack is empty, the function returns ‘0’

General functions:

int readInputFile_toCreate ( )

That function takes three parameters; first input file, stackSizesArray, queueSizesArray. Returns client_size after reading the input file. While function is running, it fills the stackSizesArray and the queueSizesArray up.

int readOperationFile ( )

That function takes two parameters; second input file and operationArray. Returns operation_size after reading the input file. While function is running, it fills the operationArray up.

void simulate ( )

That function takes seven parameters and all work is done in this function. It separates the operationArray into usable parts. This function invokes the functions needed according to the incoming operation and prints the results to a array.

3.3. Algorithm

1. Read the input files and set the necessary arrays
2. Reserve space from the memory according to the information read from the file
3. Separate submitted operations into classes
4. Do the necessary actions for separated operations
    - 4.1. Check errors for each operation
    - 4.2. If there is no error, do the post operation
5. Print the result to output file
6. Close files

3.4. Software Testing Notes

The suitability of the operators given as inputs in the program is not checked, they are only assumed to be correct.

On the other hand, even if all inputs are correct, no control is made when the process request command for the server arrives.

Finally, it is also not checked whether the values sent to the program as arguments are present. If argument does not exist, the program will not run.

4. COMMENTS

This assignment is very educational for stack/queue logic and dynamic memory management. Also pointers need to be used for structure. In a given sense this assignment comes up with the time and difficulties in a parallel way. The materials provided and the answering to each question was very good.

5. REFERENCES

- https://www.cs.bu.edu/teaching/c/stack/array/
- https://www.cs.bu.edu/teaching/c/queue/array/types.html
- https://www.tutorialspoint.com/cprogramming/c_structures.htm
- https://www.programiz.com/c-programming/c-structures