DSA Team 48 Project

Vidit Jain

May 2021

Data Structures Used

Strings

Looking at the project initially, we knew for a fact that we'd need to make use of strings very frequently, as we'd have to take in string inputs, parse, manipulate, concatenate etc. Using malloc everywhere, and having to pass the length of the string separately makes it inconvenient to code, and is bound to lead to errors. Therefore, we decided to implement a string data structure, that would store a character array, and the length of the string it holds. We wouldn't have to deal with working with character arrays directly, and allocating them memory as the functions coded by us did them for us and provided a standard. This improved the readability and simplicity of our code significantly.

Token Matrix

To deal with tokenizing the input provided to us by the user, we ended up deciding on creating another data structure for holding an array of strings, as well as the number of tokens inputted. Using this data structure, we were able to pass the tokenized input to all of the commands without facing any issues, and helped in modularizing the code as well.

The token matrix is basically an array of strings. The strings correspond to the inputs provided by the user. An array implementation was chosen over stacks as it is more compatible with the execvp function. The space used is also constant in our case. Since each argument is added after the previous argument an array implementation allows insertion in constant time, hence the time complexity is not hindered. The array implementation also make using the use function very easy since the we can access any element in the token matrix in constant time. We know which argument we want to change and hence can access it's index and change it appropriately.

For example:

```
use Assignment1
```

```
compare Assignment1.zip
```

the normal number of arguments for compare is 3 i.e.

```
compare Assignment1 Assignment1.zip
```

However, since we used `"use Assignment1"`, we need to make changes to the token matrix so that Assignment1 is added between `‘compare’` and `‘Assignment1.zip’` and then the token matrix can be passed on to the corresponding function just as it was normally. The array implementation makes this process very easy. Hence a stack implementation (or queue or priority queue) was avoided.

Processes

Our project required only one process to run at a time. When a child process was created for execvp, it was terminated before returning control to the parent process. Hence we didn't need any data structure to store the processes. However, for running multiple processes we can implement a priority queue type data structure, every time a child process is created the parent process can be pushed to the priority queue along with its priority, after the current process gets terminated the next process is popped from the priority queue in order of highest priority first.

Tree

For the `tree` command, we implicitly create a form of a tree, by showing the relation between the files and the directories and levels. This helps us understand the structure of the assignment better, as it displays the hierarchy of the files, which cannot be done elegantly using a linear data structure, hence using a tree made the most sense in representing the file structure.

Algorithms Used

Depth First Search

- The tree command creates an indented file to represent the file structure of the assignment the user enters.
- To implement this, we used `dirent.h` to scan through all the subdirectories and files of a given assignment.
- However, this only provided us the subdirectories and files at only one depth deeper than the parent, and we wanted to display the entire file structure of the assignment, including files and subdirectories at all levels of depth.
- Therefore, we felt that using depth first search would be optimal at using the function of `dirent.h` to print all the files and folders in the assignment folder.
- We coded up a recursive function, that would take a path as a given input, and iterate and print all the subdirectories and files it came across. Once it hit a folder, it would concatenate the folder name to the current path and call the function again, with an extra indent to show that the depth has increased.
- If it's a file, it doesn't call the function recursively as a file cannot contain files within.
- This provided us with a properly indented, visual representation of the file structure of the assignment.

The time complexity of this algorithm is $O(F + D)$, where F is the number of files and D is the number of directories in the assignment folder.

Two pointer method

- For the compare function, we realized that we have to not only handle if the files have been modified, but also if they have been deleted/created.
- Handling each case would lead to quite a hassle and hard-to-read code. Therefore, we thought of finding a more elegant solution in which we handle all 3 cases in one go.
- By sorting the files we have to check lexicographically, we could make use of the two pointer method.
- We'd start with the first file in both of the lists, where the first list is the list of files in the directory, and the second list is the list of files in the zipped file. The method can be simplified into a couple cases.

1. Check if the two files have the same path and name, and if they do, then compare the hash values of the two files and output a warning in case they are different, then increment both of the pointers in the lists.
2. If the file in the first list is lexicographically greater than the file in the second list, that means there is a file missing in the first list, and we increment the second list's pointer.
3. If the file in the second list is lexicographically greater than the file in the first list, that means there is a file missing in the second list, and we increment the first list's pointer.

- At the end, if we've gone through all the files of one list but there are still files left in the other, we report all those files as missing from the list.

We first need to sort the files by name, which would take $O(FlogF)$ where F is the number of files we are comparing.

Subject Array

We use the `dirent.h` header file for finding the number of files in the server. The server is accessed twice, first to count the number of files and then to store the number of files.
The loop is run twice to save space. After the number of files is counted we initialize the subject array (array of strings) to exactly the number of subjects in the server. Then we iterate over every subject and store it into the subject array. This takes linear time and depends upon the number of subjects in the server. This is a relatively cheap operation as a single student has on average between 3 to 9 courses in a semester.

Therefore the time complexity of this is $O(n)$ where n is the number of subjects.

Processing and Parsing

The time taken to process the input depends upon the number of characters entered by the user (including spaces). The tokenizing part depends upon the strtok functions and the number of arguments provided by the user. These are cheap operations as the number of arguments for any function is at most 3.
Some commands involve loops to determine whether a subject or a folder exists, but these don't affect the run time as the number of files being processed is less.
For the assignment shell most of the execution related work is done by system and execvp.

Commands

Most of the commands run in constant time as they involve making, copying and moving files and folder between the Subjects folder and the Server. Some commands, such as `create`, `update`, `submit`, `tree` have linear time complexity, depending upon the number of files in the directory.

`setup` has linear time complexity, depending on the number of lines in the `setup.txt` file.

`switch` is constant as we just have to change the directory.

`use` is also constant as we only have to store of the name of the Assignment we are storing.

`test` depends upon the time taken by the `submitter.py`

`compare` has $O(FlogF)$ time complexity, where F is the number of files present.

Division of Work

Anirudh Kaushik

processor/input

This is a header files which contains the struct declaration for the String type variable. It is a struct which contains a pointer to a character and an integer denoting the length of the string. All of the function use this struct variable. The string.c and string.h files take this struct as an input and perform the required functions on it.

processor/prompt

Prompt provides the terminal interface.

processor/prompt_init

Initializes the home_path and current_path strings which are used by several utility functions.

processor/prompt_print

Prints the username and current subject which is sent to it by sysinfo. In case the use command is active, prompt_print will also print the name of the assignment in use enclosed within brackets as: <Assignment_num>.

processor/tokenizer

Takes input from the `main.c` in the form of the struct string variable defined earlier with spaces included. This is the command entered by the user. In order to use this command efficiently along with functions such as `execvp` and `system`, the user input is tokenized using strtok functions and inputted into an array of strings which makes the use of the command and it's arguments a lot easier.

The tokenized input is stored in an array of strings in the order they were entered, for instance:

```
compare Assignment1 Assignment1.zip
```

will be stored as:

compare $\rightarrow$ arg[0]

Assignment1 $\rightarrow$ arg[1]

Assignment1.zip $\rightarrow$ arg[2]

arg[3] = NULL;

To make the tokenized input compatible with execvp the last argument is made to point to a `NULL` character. We have a struct variable named token_mat to store the tokenized input. It consists of a pointer to a pointer to a char (array of strings) and the size allocated to it is 100 (max number of tokens) and the size allocated to each string is 100 (max token length). The tokenizer then appropriately renames its output and then sends it to a function called execute defined in the same .c file.

The execute function compares the first argument in the token matrix and sends the token matrix to the corresponding function.

In the above example the first token is compare, hence it will be sent to the `commandCompare` function which takes the token matrix as an input.

All the commands take the token matrix as an input and perform the required manipulations to it. The token matrix structure was chosen to be an array of strings rather than a stack as this structure was more compatible with `execvp` than a stack and also it made input, output and processing of strings a lot easier.

commands/use

The use command is a command implemented for user convenience and allows the user to make multiple changes to a particular file without having to specify it's name each time. Every function that takes an assignment name as an arguments depends upon the use function. To ensure that the project is modular and every function can be implemented independently to the other, the use command was implemented in the tokenizer function itself since this would not require changing all the functions manually to accommodate it. The use.c file has a function to update the global variable (`UseCond`) and a function that initializes the name of the Use_Assignment_name string with the name of the subject entered by the user. The user can also choose to use a different assignment name with the use function. Inside the execute command of the tokenizer function, the number of arguments passed to a particular command is checked and if it is one less than the number of arguments required then the value of `UseCond` is checked. If `UseCond` is set to one then the function would automatically update the token matrix with the assignment name as an argument before passing it to the corresponding function.

If the number of arguments is less and the `UseCond` is set to 0 then the token matrix is passed to the functions as is and in this case the function will tell the user that the way they used the given command is wrong and it required more arguments. The use command is made more effective and easier to use by organizing the Server. Every subject contains an Assignment_num folder which has the same syntax across all subjects. This allows the user to use the use command across different subjects since they all have the same syntax thus increasing the convenience of the command.

commands/execvp

The `execvp` consists of a header file and the corresponding .c file. The `execvp` taken the token matrix as an input and creates a child process. This is done as `execvp` terminates the calling process upon execution of the command. Since we're dealing with only one parent process and only one child process and the child process gets terminated after the execution of the command is complete we decided to avoid the use of a deque or a queue to store the processes. Since the parent process and the child process are executed in parallel, it was required to put the parent process on hold till the child process is terminated. To do this we use the `usleep` function provided in `unistd.h` to delay the execution of the parent process by a fraction of a second. In this time the child process finishes executing and is terminated and control is returned to the parent process. An if condition checks whether the current process is the parent process or child process. The header file contains the required `#includes` and the function prototypes.

utils/Input_mat

The token matrix is an important struct variable used by all the commands and is declared here. Input mat.h contains the struct variable declaration and a function prototype.

Input_mat.c contains the function which creates an input matrix. This is used by tokenizer to store all the tokens inputted by the user and the struct is passed around to all the functions. This implementation allows us to extend this project to include multiple process as each tokenized input is stored separately. So for multiple processes the input to each process can be pushed on to a priority queue type data structure and the commands can be executed in non-decreasing order of priority (highest priority first). In our current implementation the matrix is discarded after its use.

utils/sysinfo

sysinfo consists of a .c and a .h file. sysinfo helps obtain the username of the use which is passed to prompt.c which in turn displays it. For example:

```
JackOfTheBean:subject_name/$
```

The username JackOfTheBean is provided by the sysinfo function.

utils/Subjects

Subjects is a function that allows us to obtain the names of all the subjects stored in the server. Every time a new subject is added to the sever, the subject matrix is initialized will all the subjects. This matrix is stored as a global variable and it is used by functions such as switch subject and files.c which obtains the current subject by comparing it with the names of the subjects stored in the subject matrix and returns only if a given subject is also present on the server. This is implemented so that every time a new subject is added to a server it is automatically obtained in the subject matrix.

Server

The server (a local directory) is where the student obtains all his assignments from and submits all his assignments to. The server contains a folder with the name of all the subjects and inside each subject is the name of the assignments.

To ensure all the commands can be used conveniently by the user the server is organized so that every assignment name has the same structure and syntax.

Every assignment can be named → Assignment_num. and must contain the dist folder specifying the folder structure of the assignment to be submitted and the `submitter.py` file.

main.c

main.c facilitates interaction with the user.

In the beginning the global variables are initialized to their corresponding starting values followed by an indefinite while loop that takes user input and sends it to the tokenizer function. The quit condition is defined within tokenizer and allows the user to exit the Assignment Shell.

General

Provided help with debugging and provided ideas for implementation of certain functions. Made some changes to get current subject function in files.c to get all the subject names in the subject matrix and compare the argument of the current subject with the subject array, returning the current subject only if it true.

Extra

`list`: list all the current files in the directory.

Vidit Jain

commands/compare

In the compare function, I would first check if the assignment and the zip file exist in the first place If even one of them don't, I would output an error message to the user

To compare, I would unzip the folder into a temporary folder. Then I would recursively find all the files in the directory of the assignment folder, sort them and perform an md5sum on all and output it to a file, and the same for the unzipped folder as well. I would then go line by line through both of the files, and output whenever a file was missing from either folder, or a file was modified. If there was no difference, I would output that the integrity has been maintained and that there is no difference between the assignment and the zip file.

commands/setup

I would first check if the setup file existed in the dist folder of the assignment. Then the code goes through the entire file and verifies if this is a valid file structure. (You can't go 2 levels deep in one step, valid file names, only one assignment being set up).

Then I would go line by line and using the number of indents on that line, I would accordingly create the folder in the right directory, storing the path in a string and changing it accordingly whenever the number of indents changed after jumping to another line.

commands/tree

Used the directory entry library to list all the subdirectories and files of a given assignment. I implemented it such that it recursively called itself and outputs the indented file structure in a text file stored in the assignment.

utils/files.c

`folderExists, fileExists`: Made use of sys/stat.h to detect if the path inputted is a directory/file respectively.
`deleteFile/deleteFolder`: Checks if the file exists and deletes it using rm and calling system.
`createZip`: Checks if the folder you're trying to zip exists, and zips it using `zip -r` on the folder through the system command.
`unzipToDirectory`: Checks if the zip file exists, unzips it to a temporary folder using `unzip -r` through the system command.
`createFolder`: Creates a folder using `mkdir` through the system command if the folder doesn't already exist.
`validFileName`: Uses regex to decide whether the entered string is a valid name for a file/folder (more restrictive than most OSes, intentional).
`countLines`: Opens a file, and iterates through every line to count how many lines are in the file.

General

Led the Github Repository, reviewed all pull requests, helped modularize code to make everyone's files work with each other seamlessly and set a platform so that each person can contribute their part independently without any confusion. There were a few merge conflicts that had arisen, which I reviewed and fixed.

Ayan Agarwal

utils/string

1. It has a struct which consists of a string (pointer to char) and length of the string.
2. It has many functions which are basically present in the string library of C, but this ADT was created to access the strings more easily and we stored the length of the string beforehand so that it doesn't need to get computed again and again.
3. Also, along with copy string, concatenate 2 strings (attach string) and compare 2 strings functions, it also has many functions such as make empty string, make string, break string and delete string. These functions basically increase the functionalities of the code as these functions involve malloc data to struct variable, calculating string length and storing it in struct string variable.
4. This ADT was used throughout the project and its functions were used at various steps of code.

commands/submit

1. When this function is called by command `submit <assignment_name>`, it firsts check whether you are in a subject directory or not and gives error if not. Now, it checks if that assignment folder exists in the present directory (subject here, It uses the InSubject global variable to check if we are currently in a subject or not), if the zip file already exists, it deletes it and then, creates the zip file for that assignment. Now, if the zip file is not created it gives an error because the assignment folder is not present, otherwise it returns successfully after creating the zip file.

2. The submit part basically checks if zip exists there at first, if it doesn't, then it returns with an error message. Else it goes into submitting the zip file of the assignment onto server. It checks if there is a submission folder or not under the respective course's folder in server folder, if not, it creates one.

3. Now, it checks if the zip file has already been submitted onto server in submissions folder under respective course's folder, if yes then it asks us whether we would like to "Overwrite" it or "Return" without doing anything. If it doesn't exist in submissions folder (server) then it just copies zip file there and returns.

4. This function has kept care of the edge cases at all the places possible and provides users with proper information about it and prompts to correct it.

utils/files

It has all the necessary functions needed to create folder, check file's/folder's existence, creating zip, copying(submitting) the zip file onto server, creating submission folder,etc. The code for files.c was written by me which involved functions related to submission on the server and creating submission folder, etc., basically functions that were needed for main functionality of submit command and some supporting functions. Also, it is being given input(assignment_name) and called by tokenizer function. It uses the InSubject global variable to check if we are currently in a subject or not.

Sreejan Patel

commands/switch

1. The Syntax of switch command: `switch <subject>`.

2. The switch command is called by the tokenizer function with input being the subject name.

3. The switch command uses files.c, string.c, sysinfo.c and globals.c for its implementation.

4. As we enter the command to switch into a subject , the switch command first checks if we are already in any subject i.e the `isInSubject` , which is a global variable , is 1 if we are already in a subject or is 0 if not .
5. If we are already in any subject we move back to the previous directory , where all the subjects are present.
6. Then it checks if the subject entered by the user is present in the local machine.
7. If there is no such subject and if the `isInSubject` is 1 , then moves into the subject directory which we were in at the start.
8. If the subject exists, then we move into the subject directory which was entered by the user and update the `isInSubject` value to 1.
9. This function takes care of all the edge cases at all the places possible and works as intended by the user.

commands/update

1. The syntax of update command: `update <assignment>`
2. The update command is called by the tokenizer function with input being the assignment name.
3. The update command uses files.c, string.c,create.c and globals.c for its implementation.
4. First it checks if we are in a subject or not. If we are not in any subject it returns error , since the assignment will be located inside the subjects.
5. If we are in a subject , then it checks if the assignment entered by the user is present in the local machine or not.
6. If the assignment is not present in the local machine but present in the server i.e it is not downloaded yet , then we call the createassignment function present in the create command to create the assignment in the local machine.
7. If the assignment exists in the local machine then , first we check all the pdf files that are present in the server assignment and delete only those files since the other files can be that of the user and we also delete the dist folder.
8. After deletion we copy the dist folder and the pdf files from the server to the local machine assignment folder.
9. This function takes care of all the edge cases possible and works as intended by the user.

Yederu Rupasree

commands/create

1. commandCreate takes 'argument args_mat ' which is a struct containing array of arguments to a command to be implemented and no of arguments made by 'tokenizer'.

2. The function first checks whether no of arguments is one(which should be name of assignment).If not,it will print error message to enter valid command.If yes,it checks whether we are present in a subject directory or not using "`isInSubject`" function and prints appropriate error message to switch into a subject.

3. if yes,it should call function create which needs arguments Serverpath (including subject) and assignment name.

4. To get assignmentname make a string "args_mat.args[1]" which is basically assignment name entered by user.To get serverpath make a string ("../../Server") and attach it name of subject using 'getCurrentSubject()'.And pass these arguments to call createassignment function.

5. It takes input from 'tokenizer',uses '`isInSubject`'(global variable),'getCurrentSubject' in utils/files.c,uses 'make_string','attach_string' in utils/string.c

6. Createassignment takes String* serverpath,String* assignment as arguments.As serverpath contains path to a subject folder we need to create a new String attaching '/' and assignment name(assignment->str) to server path to get path to assignment folder in server (which is "folder" here.)

7. First it checks whether assignment folder exists on sever or not using "folderExists" which takes argument folder and prints an error message if not.

8. If yes,it checks whether assignment is already created or not.If not,it creates assignment folder using 'createFolder(String* Folder)' sending assignment as argument.

9. Then,it needs to copy dist and problemstatement from server.first it makes String* dist to store path to dist atttaching "/dist" to "folder->str"(path to assign on server).

10. To copy dist folder we need to create a command that can be send as argument to system function.Make an empty String* and write command->str as " cp-r dist->str assignment->str" using sprintf and send it as argument to system() to copy dist from server to assignment.

11. As problem Statement can be pdf with any name we need to find files with .pdf extension in assignment on server and copy it to assigment.this can be done using "find sourcedir/ -name "*.pdf' -exec cp -pr '' 'destdir/' ';' "

.this command finds files with .pdf in source directory and sends them to as arguments to exec function which then copies these files to destination directory executing cp command.

12. After copying both files assignment will be created.

13. It takes input from commandCreate,uses ‘folderExists’,’fileExists’,’createFolder’ in utils/files.c, uses ‘make_string’,’attach_string’ in utils/string.c

commands/test

1. `commandTest` takes argument `args_mat` which is a struct containing array of arguments to a command to be implemented and no of arguments made by ‘tokenizer’ .

2. The function first checks whether no of arguments are one(which should be name of assignment).If not,it will print error message to enter valid command.If yes,it checks whether we are present in a subject directory or not using `isInSubject` function and prints appropriate error message to switch into a subject.

3. if yes,it should call function test.To pass arguments to create function assignment name.To get assignment name make a String `assignmentName` `args_mat.args[1]` which is basically assignment name entered by user. And pass *assignmentName as argument to `test` function calling it to test the assignment.

4. It takes input from ‘tokenizer’,uses `isInSubject` (global variable) in `utils/files.c`,uses `make_string`,`attach_string` in `utils/string.c`

5. test takes argument String folder which contains `assignmentname`. First it makes a `String *file` which stores path to `submitter.py` attaching `dist/submitter.py` .Then it checks assignment and `submitter.py` exists or not using

6. `olderExists` and `fileExists` prints appropriate error message(if not).

7. If both are present it needs to run `submitter.py` and store it to Logs first. It checks if Logs folder exists using `folderExists`.If not creates a Logs folder using `createFolder`.

8. Logs may contain multiple log files,so it needs to check if i-1.log already exists and i.log doesn’t exist starting from 1.log we redirect the output to i.log’It runs the submitter.py and store the output in Logs/i.log.

9. It uses ‘folderExists’,‘fileExists’, ‘createFolder’ in utils/files.c, uses ‘make_string’, ’attach_string’ in utils/string.c

Running MOSS

To be able to answer this question we must first consider how our server functions and communicates with the main server.

We have assumed our server to work like Moodle. Each student can access only their specific part of the server which is updated with global resources from the main server (such as problem statement, dist folder etc.)

As a student can submit any file with any name on Moodle, a student can submit a folder without their roll no. to the server. The constraints of the project prevent us from renaming the folder to anything other than Assignment_num(Assignment1, Assignment2,etc.) without significant loss in efficiency.

When the server uploads the students submission, the TA will be able to view it under the students email id registered with the institution. So the server uploads file to the main server in the repository allocated to the student associated with his/her/their id. This allows Assignment submissions with the same names to be distinguishable.

Now on to making running of MOSS more efficient and easy. MOSS requires a pair wise comparison of each submission and hence there are always nC2 comparisons for n submissions. Thus to make this process easy, Our server will upload files to the corresponding repository of the student in the main server. All the submissions corresponding to a particular assignment will be put together in one folder and a MOSS search can be run for each file along with the rest of the files on the server.

To make life easier, we could implement a command

```
moss <submissionsFolderName> <percentage>
```

that runs MOSS on all the subfolder in the specified submissions folder, and outputs the pair of people with high percentage similarity, with the percentage being decidable by the person running it. After a MOSS call is done between two people, the program can open the link given and parse the site to get the MOSS percentage, and outputting it to a csv file, which can then be opened in Excel or Google Sheets for simplicity and ease of use. once a particular student's submission has been scanned he will not be checked again to reduce unnecessary comparisons.

In order to be able to recognize repeat offenders and groups of cheater, copy cases, we can use a graph: An edge will be formed if a MOSS match between a pair is more than 50% (as anything less may be a false positive). Then we can divide the graph into components to see which group copied the code from the same source, by using disjoint set union. The graph can be retained for comparisons in the future.

Thus our Assignment shell allows MOSS to be implemented in an easy and efficient manner.