GTU

CSE344 - System Programming

Homework1 - Report

Ahmet Tuğkan Ayhan

1901042692

For the homework, I created my own libraries and wrote my functions inside these libraries. Because otherwise there would be a lot of mess inside main.c file. For file operations I used :

- **File.h**
- **File.c**

And for parsing operations I used :

- **Parser.h**
- **Parser.c**

Let's start explaining the code with main.c file :

<u>Main.c</u>

```c
int main(int argc, char * argv[]) {
    /* INITIALIZING */
    controlArguments(argc, argv);

    // Getting str1 and str2 from command line arguments
    char argumentStr[MAX_STR_LEN]; strcpy(argumentStr, argv[1]);
    // Initializing File Path
    char inputFilePath[MAX_PATH_LEN]; strcpy(inputFilePath, argv[2]);
    // Getting length of the file
    int  fileLength = getFileLength(inputFilePath);
    // Allocating enough space for fileContext(gave extra space because of replacement)
    char fileContext[MAX_STR_LEN];

    // Reading context from the file
    readFromFile(inputFilePath, fileContext, fileLength);
    // Changing occurrences
    changeAllOccurences(argumentStr, fileContext);
    // Writing new context to the file
    writeToFile(inputFilePath, fileContext);

    return 0;
}
```

Procedure:

- First, checks if the command line arguments are correct or not

- If arguments are correct then assigns these arguments to local variables
- After that, stores length of the file with **getFileLength** function which uses lseek to move inside the file.
- Then, sends these informations to **readFromFile** function(which is a part of **File.h** library) and stores the file context inside **fileContext** variable
- Changes all occurrences with a function from **Parse.h**
- Finally, writes edited file context back to **inputFilePath**

File.h

```c
#ifndef FILE_H
#define FILE_H

#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int  getFileLength       (char * filePath);
int  openFile            (char * filePath, char * fileType);
int  readFromFile        (char * filePath, char * fileContext, int fileLength);
int  writeToFile         (char * filePath, char* fileContext);
void closeFile           (int fileDescriptor);
void lockFileForWrite    (int fileDescriptor, struct flock * lockStruct);
void unlockFileForWrite  (int fileDescriptor, struct flock * lockStruct);

#endif
```

getFileLength

- First, calls **openFile** function
- After getting file descriptor, sets starting point of the file to the end of the file and assigns this value to length of the file
- Sets starting address back to normal(to start) and closes the file

- Finally returns the file length

<u>**openFile**</u>

- Checks file type. For example, if given type is “r” then opens the file with read flag and assigns the file descriptor value
- After getting file descriptor value, controls if it is valid or not
- If it is valid, then returns the file descriptor value or exits after showing the error

<u>**readFromFile**</u>

- Calls **openFile** function with “r” flag.
- Since file length is already known(given as parameter), calls **read** function with that length and stores context inside **fileContext** parameter
- Before closing the file, checks for possible errors by calling private **checkErrors** function.
- Finally, if there is no error, returns the amount of bytes read from the file.

<u>**writeToFile**</u>

- First, creates a lock to prevent multiple write operations on the same file at the same time.
- Then, calls **openFile** function with “w” flag
- Since how many bytes are going to be written is already known, calls **write** function with these values.
- After assigning amount of bytes written to length variable, calls **checkErrors** with this variable and controls any error.
- If there is no error, unlocks the file and closes the file descriptor
- Finally, returns the amount of bytes written to the file.

<u>**closeFile**</u>

- First, closes file by calling **close.**
- Then, by using the return value, calls **checkErrors** function to check if there is any error. Exits the code if there is an error after printing out the cause.

<u>lockFileForWrite</u>

- I used lock system that is available in the **week3.pdf** of our lecture.
- It first sets the lock type to **F_WRLCK**
- Then, calls the **fcntl** function with necessary parameters

<u>unlockFileForWrite</u>

- It acts very same as **lockFileForWrite** function but this time it sets the lock type to **F_UNLCK**
- Then, calls the **fnctl** function.

Continues...

Parser.h

```
#ifndef PARSER_H
#define PARSER_H

#define MAX_STR_LEN 16384
#define MAX_PATH_LEN 1024

#define CASE_SENSITIVE 0
#define CASE_INSENSITIVE 1
#define ONLY_START 1
#define ONLY_END 1

struct Strings
{
    // Base Str
    char baseStrings[100][MAX_STR_LEN];
    int  numOfBaseStr;
    // Target Str
    char targetString[MAX_STR_LEN];
    // Parser features
    int caseSensitivity;
    int onlyStart;
    int onlyEnd;
    int repeatLimit;
};
struct Strings getArgStructure(char rawStr[], char * fileContext);

int   changeOccurrence    (char* str, char * delimiter, struct Strings * argStruct);
void  changeAllOccurences (char * argumentStr, char * fileContext);
char* myStrlwr            (char * str);
char* myStristr           (char *__haystack, char *__needle);
void  getPart1            (char * str, char * part1, char * delimiter, char *
firstOccurence);
void  getPart2            (char * str, char * part2, char * delimiter, char *
firstOccurence);


#endif
```

#define

- For the define part, I used 6 different define rules.
- **MAX_STR_LEN:** represents the maximum length limit for context inside of the .txt file
- **MAX_PATH_LEN:** represents the maximum length limit for command line argument -> **inputFilePath**
- **CASE_SENSITIVE:** used inside **Strings** struct. If **caseSensitivity** equals to **CASE_SENSITIVE** it means there is no **'i'** character inside command line argument
- **CASE_INSENSITIVE:** used inside **Strings** struct. If **caseSensitivity** equals to **CASE_INSENSITIVE** it means there is an **'i'** character inside command line argument
- **ONLY_START:** The variable **onlyStart** inside Strings struct is only assigned to **ONLY_START** if there is a **"^"** character inside cla.
- **ONLY_END:** The variable **onlyEnd** inside Strings struct is only assigned to **ONLY_END** if there is a **"$"** character inside cla.

Strings Struct

- **baseStrings** simply holds all possible word cases that is going to replaced with **targetString.** In other words, any string inside **baseStrings** will be replaced with **targetString** after **changeAllOccurrences** is called.
- Size of the **baseStrings** is hold with **numOfBaseStr**
- **targetString,** is the string that is showed as str2 inside the homework instruction pdf.
- **caseSensivity(i), onlyStart(^) and onlyEnd($)** are used when any character related to them occur inside cla.
- **repeatLimit** represent limit for the star symbol. For example if str1 is given as "elm*a" and repeat limit is 5, then the maximum replacement will occur at elmmmmma. It will replace any word that contains 5 or lower amount of m. But initially I gave the limit for it as 20 which is not likely to exist.

getArgStructure

- It is first called by the **changeAllOccurrences**, which is the only parse.h library function used in main.c.
- It simply gets a **rawStr**, which is cla (for ex: /str1/str2/), and processes that rawStr to fill all variables inside **Strings** structure.
- Before doing anything, it creates a **Strings** struct and initializes some variables of it. (for ex: repeatLimit = 20)
- Then, it calls **controlSemiColumn**, **controlUpperArrowDollarSign**, **controlSquareBrackets**, **controlSensivity** and finally **controlStars** in this order. Every time one of these functions called it creates possible str1 cases and assigns them to the **baseStrings** variable inside **Strings** struct.
- Finally, after getting all possible word cases (for ex: str1 sttr1 StR1 and so on) and target value (for ex: str2) it returns the **Strings** struct.

changeOccurrence

- It is called by the **changeAllOccurrences** function
- It's job is to replace str1 with str2. To do this, first it gets the str as the parameter(which is all the file context)
- Then it searches given delimiter(str2 in this case) inside str and divides the str into to parts; part1 and part2. To make it clear we can think this example:
  str: "Bugün str1 beni ziyaret etti."
  delimiter: "str1"
  part1: "Bugün "
  part2: " beni ziyaret etti"
- After dividing is complete, it replaces str1 with str2 and appends the parts back together.
- What is different is, it doesn't replace all occurrences. I mean, if there is more than one str1 it will only replace the first one. To make it work, changeAllOccurrences calls this function until it returns 0
- If there is still str1 occurrences, then it will return 1

changeAllOccurrences

- It abstracts everything from the user.
- First calls getArgStruct, and assigns the return value to local struct variable.
- After filling the struct it calls **changeOccurrence** until everything inside **baseStrings** is replaced with **targetString**.

myStrlwr

- This is a helper function that I wrote for myself.
- It simply takes a char pointer as a parameter and it turns every character inside that char pointer to lower character (for ex: A->a, B->b, 5->5)

myStristr

- This is an edited version of the strstr function inside string.h library.
- Normally strstr function returns first occurrence address of the substring inside a string.
- But unfortunately this string.h library function is case sensitive. To use it with case insensitive option, I wrote the same library function but case insensitively.

getPart1

- This function is used inside **changeOccurrence** function. To remind the example, we had:
  str: "Bugün str1 beni ziyaret etti."
  delimiter: "str1"
  part1: "Bugün "
  part2: " beni ziyaret etti"
- Part1 of this example is calculated by this function

getPart2

- Same functionality as getPart1, this function calculates the part2.

- How it works is, since we know str1 first occurrence address with strstr function, if we add delimiter(str2) length to this occurrence address we get the part2 starting point.
- After getting starting point, parameter “part2” given with function call is filled character by character.

- END -