Assignment 2 : Word Wrapping

Written by : Alvin Zheng

NetID : az376

0. Background Knowledge

> What is a buffer: a buffer is a region of memory that temporarily stores data before being flushed out. Here in this project, the buffer can also be known one line, where the buffer length is the size of the line. The only exception is when there exists a word that is longer than the maximum length of the buffer.

1. Overview

> What this `Word Wrapping Machine` does is given a piece of text, column width, it reorgainizes the text into a satisfied width. It also eliminates any extra spaces and extra lines.
>
> Ways of Operation:

- Reading from standard input
  - First Step: Type "Make"
  - Second Step: Type "./ww *width*" (where width is a positive integer)
  - Third Step: Hit the enter key
- Reading from a text file
  - First Step: Type "Make"
  - Second Step: Type "./ww *width text.txt*" (where width is a positive integer and text is the name of the file)
  - Third Step: Hit the enter key
- Reading all the text files in the directory
  - First Step: Type "Make"
  - Second Step: Type "./ww *width text*" (where width is a positive integer and text is the name of the directory)
  - Third Step: Hit the enter key
- Reading from all text files in the directory with subdirectories
  - First Step: Type "Make"
  - Second Step : Type "./ww -rN,M 20 directory" (where N is number of directory threads and M is number of file threads, directory is the name of the starting directory)

* Third Step: Hit the enter key

2. Files/Directories

> You are given four sample text files to run with. Feel free to add more into the "text" directory. Make sure to place the file into the correct directory and `DO NOT` delete this directory or else errors may occur.
>
> You will also see a directory called SystemData. `Warning:` DO NOT delete this directory or else errors may occur.
>
> ww.c is the source code, ww.h is the header file.

---

3. How the machine operates

`The Temporary Array to store single words` :

```
char *temp_array;
```

This array stores the chrarcters of the current word being read. When we encounter a space or a new line, it indicates that the word come to a end. This is when the word will be transfered to the buffer accordingly.

```
int word_length;
```

The length of the temporary array

`The Buffer` :

```
char *buffer;
int max_length;
int char_read;
```

max length is the maximum length of the buffer, and the temporary array,. char_read is the current index of the buffer, which will be zero when the buffer is emptied out.

`Initializing and Finalizing Functions` :

```
struct HarmonyOS * initialize(char *size);
void finalize(struct HarmonyOS  * localData);
```

Initialize() will allocate memory on the heap for the buffer and temp_array, and finalize will free them

`File Descriptors` :

```
int file_read;
int file_write;
```

file_read is the file descriptor of the file being read, or input file

file_write is the file descriptor of the file being written to or output file

`Useful Functions and their brief explanation` :

```
int checkDirectory(const char * path);
int checkFile(const char * path);
int special_characters(char *ptr);
void print(char *array, int length, int mode, int type,struct HarmonyOS  * localData);
void print_word(int type,struct HarmonyOS  * localData);
void transfer(int mode, int type,struct HarmonyOS  * localData);
void empty_buffer(int mode, int type,struct HarmonyOS  * localData);
int isEmpty(char *array, int size);

void throwError(const char *path);
int checkTXT(char *name, int length);
int validness(struct dirent *file, char *directory_name, char *str);
void addTwoStrings(char *one, char *str, char *two, char *result);
void readFromDirectory(char *directory_name,struct HarmonyOS  * localData);
```

> **checkDirectory()**: Checks whether the file is a directory exists under the current path
>
> **checkFile()**: Checks whether the regular file and whether it exists under the current path
>
> **special_characters()**: checks whether the given character is a special character such as a space or a new line
>
> **print()**: print out the entire given array to a specificed output source given by the parameter "type"
>
> **print_word():** prints out the REMAINING of the word (until a space is detected) to a specified output source given by the parameter "type". This function is usually called when a single word is larger than the maximum width.
>
> **transfer()**: transfers all the characters in the temporary array (also known as the word array) into the buffer
>
> **empty_buffer()**: empties out the buffer and print the characters into a specified output source given by the parameter "type"
>
> **isEmpty()**: checks whether a given array is empty
>
> **throwError()**: given the path to the error, throw error calls perror() and `terminates` the program
>
> **checkTXT()**: checks whether the file names ends with a '.txt'. this is ONE of the multiple checks that are performed to check if the file is a text file
>
> **validness()**: a helper method for readFromDirectory(). It checks if the file has >4 characters since '.txt' is already 4 characters, whether the file is a reular file and exists, whether the file ends with a .txt
>
> **readFromDirectory()**: given the directory, it finds text files and create a wrapped version of the text file. For example, is the name of the text file is `HuoZhuoCaiYingWenJieFangTaiWanDao.txt` , it wraps that file into the desired length and prints the result to a new text file called `wrap.HuoZhuoCaiYingWenJieFangTaiWanDao.txt` ". All other files that are non text files will be ignored.

`Modes and Types` :

> mode 0: print without a new line at the end
>
> mode 1: print with a new line at the end how this is useful:

```
 if it is a words that are regular, when we empty out buffer we want a new line at the end of the sentence.
however, if we are reading a super long word that is longer than max length, we need to empty out the buffer
then and continue reading, which doesn't require a new line since it is not a sentence, it is still the same word

Your might ask : Why does "transfer", a function that transfers a singe word (stored in temp_array) into the buffer, need mode,
where it decides whether to put a ' n' and the end? Because, transfer calls the function "print", which requires mode.
print is called either from: readFromFile() or Transfer()
```

> type 1: any print statements will be towards standard output
>
> type 0: any print statements will be towards a specific file

`ReadFrom Functions`:

```
void readFromFile(char *path,struct HarmonyOS  * localData);
void readFromFileOutputToFile(char *path, char *output_path,struct HarmonyOS  * localData);
```

> The following are "readFrom" functions. they are different but have mostly the same code
>
> 1. readFromFile : Reading from given file and print to standard input
> 2. readFromOutputToFile : Reading from a given file 'A' and output to a given file 'B'.

---

4. Overall process of how the word wrapping machine wraps the text

> if the user compiles the program and runs it with the command : `./ww 20 text.txt`, the program runs with the maximum width set to 20, searches for `text.txt` under the current directory and if it is found, it wraps it to the desired length and prints the output to standard output. if it does not find the file, it will exit the program with an error:

```
./ww 20 DoesNotExist

DoesNotExist: No such file or directory
```

> Therefore, the correct way to run the program is `./ww 20 text/file.txt` where `text` is the correct directory where the text files should exist.
>
> If the User runs in a way like `./ww 20 directory`, the program will run with the maximum width set the 20, seaches for the directory `directory` and if it is found, it wraps all the text file in the directory and creates a seperate text file for each of their outputs. For example, if a text file is called `one.txt`, it will wrap the text file to the desired length and prints its results in a new file called `wrap.one.txt`.

```
$ ./ww 30 text
---------------------------------------
Program Finishes NORMALLY with Status = |SUCCESS|
NOTE: Failure means there is a word longer than maximum length characters, success means there isnt
---------------------------------------
```

> The output will be found in the directory called `output`. Notice, if the user runs this same command multiple times, it is not going to create new files or refuse to run. For saftey purposes, it replaces the existing file and create a new one.
>
> if the user inputs `./ww 20`, without a given directory or file, but a length, it is going to assume the user will input to standard input. It is first going to read from standard input and print to a new directory called `SystemData` with the file inside called `data.txt`. `WARNING`: Do not delete or modify any of these two. Serious errors will occur. Leave them alone. Then, it is going to read from the file and print to standard output. Technially, the file `data.txt` already contains the wrapped data, it is just readFromFile() that is transfering the data to the standard output.
>
> if the user only inputs `./ww` the program will refuse to run, and give the follow:

```
Check your inputs. Please give a maximum length, and if possible, the file or directory
```

---

5. Core Algorithm

How does the two main alogrithms,

```
void readFromFile(char *path);
void readFromFileOutputToFile(char *path, char *output_path);
```

run?

> They are mostly the same, the only difference is that one outputs to standard input when required to do so, and one outputs to a given file when required to do so. They both first perform their own saftey checks, and if the checks are passed, they both start reading one byte at a time until no byte is read. Each byte read require the increase of one in the `int word_length`. This is used to caculate the length of a word. Notice, any character will be counted as a word, until it reaches a space or a new line. That includes commas, periods, question marks, etc. When it reaches a space or a new line, it is going to remove that from the `temp_array` and decrease the word length by one (since it initally read every character and stores it in the temporary array). Then, it is going to transfer all the characters in the temporary array into the buffer using `tranfer()`. Notice that two new line indicate the start of a paragraph and any continuous additional new lines after two will be considered excess new lines and will not be printed. If the word does not fit into the current buffer, transfer() will empty the buffer out then put it in. If we reach a word that is larger than the maximum length, it is going to transfer the current characters to the buffer without a new line at then end then call a function called `print_word()` to print out the rest of the characters (this should be on a seperate line by itself now), then set the status flag to FAILURE. The status flag is going to be print out at the end to indicate the status of exit.

```
Program Finishes NORMALLY with Status = |SUCCESS|
NOTE: Failure means there is a word longer than "x" characters, success means there isn't
```

5. How to run

> given the make file and some sample text files, you can run by following the directions below:

```
1. compile: make
2. run: ./ww 20 text
```

> what this does is compile and run with length set to "20" characters, reading from directory called "text" and output the wrapped files to "output" .
>
> To clear the 'output' directory (although you DON'T need to since it delete and create a new file when reaching a file with the same name, such as wrapping the same file twice with different widths) use the following command :

```
make clean
```

6. Using Threads and the how HarmonyOS handles wrapping multiple files concurrently

> As you read above, you may have seen this passed into the functions.

```
HarmonyOS * localData
```

> as you can identify, localData stands for the local Data for the file, rather than shared data across threads and other files for cross data errors to occur.
>
> HarmonyOS is a strong System that allow threads to wrap files at the same time. For example, as you may have read above, to wrap a file, two main supports needed are the buffer to store the sentence, the temp_array to store a single word. If multiple threads run at the same time, they are all going to access the buffer and the word array which would just corrupt the buffer and mix up the word written in the files, since they would all have access to the same buffer and the same word array.
>
> However, with the all new data storage named `HarmonyOS`, a typedef struct that includes the buffer, word array (named temp_array in the code to temporarily store single words), and so much more, it solves the problem.

```
typedef struct HarmonyOS{
    // THE TEMPORARY ARRAY: STORES THE CURRENT WORD
    /**
     * @brief
     * temporary array to store a word
     */
    char *temp_array;
    /**
     * @brief
     * the length of the word that is currently being read.
     */
    int word_length;

    // THE BUFFER: WORDS ARE MOVED FROM TEMPORARY ARRAY TO BUFFER
    /**
     * @brief
     * "buffer" an array of characters where we store the written characters in a line
     * if the buffer is full, or cannot fit a word, we skip a line, then clear out
     * the buffer
     */
    char *buffer;

    /**
     * @brief
     * "max_length" an integer that specifies the maximum amount of characters a single line could fit,
     * also called the buffer length
     */
    int max_length;

    /**
     * @brief
     * Number of characters that currently lies in the buffer
     */
    int char_read;

    // THE STORAGE: WORDS THAT ARE READY TO ERECT INTO STANDARD OUTPUT
    /**
     * @brief
     * a place where characters go before uniformly erected into standard output
     */
    char *storage;
    int storage_index;

    /**
     * @brief
     * file_read is the file descriptor of the file being read, or input file
     */
    int file_read;
    /**
     * @brief
     * file_write is the file descriptor of the file being written to or output file
     */
    int file_write;
}HarmonyOS;
```

> With worker threads, each file will run on their own HarmonyOS, that is, their own buffer and their own temp_array (stores a single word).
>
> Without worker threads, that is wrapping only in the main thread, if we call something like "./ww 20 directory", all the files in the directory will operate on the same HarmonyOS
>
> HarmonyOS comes must be initialized and finalized and passed to functions (usually -> HarmonyOS * localData) in order for functions to operate in the correct buffer.
>
> NOTE: each file will have their own HarmonyOS (file queue node). When the file is done wrapping, the HarmonyOS will exit. It is also okay let each thread have their own HarmonyOS (since the functions will make sure that after wrapping the buffer,etc. will be empty), however is not implemented here.

7. Queues and Threads

```
//THE THREADS:
//initially all threads are 1 under recursive (-r) mode. If recursive mode is NOT activated, we will use a SINGLE thread to do readi
int threads_read=1;//number of threads used to read directory files
int threads_wrap=1;//number of threads to wrap regular files
typedef struct Thread {
    pthread_t tid;
}Thread;
int active_directory_threads;
int active_file_threads;
Thread * initializeThread();
void finalizeThreads(Thread ** array, int size);

Thread ** readThreadsArray;
Thread ** wrapThreadsArray;
```

> Directory Threads : Within the directory given by the Directory Queue, it places subdirectories back into the queue and puts regular text files into the File Queue.
>
> File Threads : Wraps the given file from File Queue
>
> there is a record of how many directory threads and File Threads. Note: All threads may be idling, which would make the number of active threads to be zero, which would end up terminating.
>
> the arrays are to keep track of the threads.

```
//THE DIRECTORY QUEUE:
typedef struct DirectoryNode{
    // A directory name
    char * directory;
    //the pointer to the next node
    struct DirectoryNode * next;
}DirectoryNode;
struct DirectoryNode * createDirectoryNode();
typedef struct DirectoryQueue{
    //the linkedlist head of the directory(s)
    DirectoryNode * head;
    // the lock that gives exclusive access to the queue
    pthread_mutex_t lock;

}DirectoryQueue;

void d_insert(char * directory, DirectoryQueue * queue);
char * d_delete(DirectoryQueue *queue);

//Global Variable of the Directory Queue
DirectoryQueue * DirQueue ;
```

```
//THE FILE QUEUE:
typedef struct FileNode{
    // A file number
    char * location;
    char * outputName;
    //the pointer to the next node
    struct FileNode * next;
}FileNode;
struct FileNode * createFileQueueNode();
typedef struct FileQueue{
    //the linkedlist head of the file nodes
    FileNode * head;
    // the lock that gives exclusive access to the queue
    pthread_mutex_t lock;
    //length of the entire queue
    int length;
}FileQueue;

void f_insert(char * location, char * outputName,FileQueue * queue);
char ** f_delete(FileQueue *queue);
//Global Variable of the File Queue
FileQueue * FilQueue ;
```

> All queues are unbounded and synchronized, means that the queue can have unlimited space, if your device allows.

8. Workers functions and how "-r" mode operates

> each thread works with its own worker. For example, directory threads work with dirWorker() and file threads work with filWorker(). Directory worker will run until there is no more directories left in the Directory queue, and the File worker will run until there is no more directories left in Directory Queue and no more files left in File Queue. Then the threads would terminate since the workers return, and go back to the main thread that is waiting for them.

```
void * dirWorker(void * arg);
void * filWorker(void * arg);
```

> recursiveWrap() is the function called when "-r" is found in the argument given. This will initialize the queues, the threads, wait for the threads to return, then finalize the queues and return back to the main function.

```
void recursiveWrap(char * directory);
```

> For more information about the user interface, please see write up!
>
> Here is an example of what should exist in the standard output (inserting/deleting order may vary)

```
$ ./ww -r10,10 30 directory
---------------------------------------
inserting text into DQ
deleting text from DQ
inserting text/empty to DQ
inserting text/text1 to DQ
inserting text/one.txt to FQ
deleting text/text1 from DQ
inserting text/text1/text3 to DQ
inserting text/text1/text2 to DQ
inserting text/text1/three.txt to FQ
deleting text/text1/text2 from DQ
inserting text/text1/text2/five.txt to FQ
deleting text/text1/text3 from DQ
inserting text/text1/text3/normal20.txt to FQ
deleting text/empty from DQ
deleting text/text1/text3/normal20.txt from FQ
deleting text/text1/three.txt from FQ
deleting text/one.txt from FQ
deleting text/text1/text2/five.txt from FQ
---------------------------------------
Program Finishes NORMALLY with Status = |FAILURE|
NOTE: Failure means there is a word longer than maximum length characters, success means there isnt
---------------------------------------
```