Operating Systems Project #3

Musa Meriç 150119751

Ayberk Türksoy 150119919

Yasin Tuğra Taş 150120048

Main Function:

Values entered as input (name of text file and amount of threads to be created) are saved in variables.(“input_file” and values[4])

The total number of lines in the file is saved in the line_count global variable.

According to the number of lines, space is created in the "writeLineArray" "replaceWorksOn" "upperWorksOn" "upper_replace_WorkDone" arrays. (What these arrays do is explained below)

```
writeLineArray = malloc(line_count * sizeof(char *));
replaceWorksOn = malloc(line_count * sizeof(int *));
upperWorksOn = malloc(line_count * sizeof(int *));
upper_replace_WorkDone = malloc(line_count * sizeof(int *));
```

Mutexes are initialized with the “pthread_mutex_init” command.

```
pthread_mutex_init(&writeMutex, NULL);
pthread_mutex_init(&upperReplaceMutex, NULL);
pthread_mutex_init(&writeToFileMutex, NULL);
pthread_mutex_init(&printTerminalMutex, NULL);
pthread_mutex_init(&readArrayMutex, NULL);
pthread_mutex_init(&readDoneArrayMutex, NULL);
```

Threads are created and directed to related functions and connected to each other with the "pthread_join" command.

Threads:

Read Threads:

When Read Threads are created, they are sent to the "GetLineNumber" method, where they get the information of which line to read from the text file and are sent to the "read_lines_from_file" method.

In the "read_lines_from_file" method, they read the text on the desired line and open the required field with the malloc function inside “writeLineArray” array and save the text here. After saving the sentence in the Array, they are sent back to the "GetLineNumber" method and wait to read another line.

Thanks to the line of code in the "GetLineNumber" method, if all lines in the text file have been read, Read Threads self-destruct with the pthread_exit method.

```
if (currentWriteLine == line_count)
{
    pthread_mutex_unlock(&writeMutex);
    pthread_exit(NULL);
}
```

Upper & Replace Threads:

When Upper and Replace threads are created, they are sent to the method named "UpperandReplaceGetWorkNumber" and they get the information about which line they will edit in the "writeLineArray" array.

We have 2 different integer arrays named “upperWorksOn” and “replaceWorksOn” to record which lines the upper or replace thread has finished.

The values in these arrays represent:

0 = has not started processing yet

1 = one thread is running on this line

2 = thread finished its process on this line

For example : For line 5 , upperWorksOn[5] = 1 and replaceWorksOn[5] = 2. This means replace thread has finished its job on line 5 and upper thread is still processing.

The reason we keep this information is that if one of the replace or upper threads is running for any line, the other threads cannot operate on this line. We check this situation in the "UpperandReplaceGetWorkNumber" method.

Threads self-destruct when all lines are edited.

Write Threads :

In order to start the process in the write threads, the replace and upper threads must be finished on any line. For this, we use the "upper_replace_WorkDone" integer array.

The values in the "upper_replace_WorkDone" array mean:

0 = not yet read

1 = The value was read from the text file, but the replace or upper threads did not act on this line.

3 = Only one of the replace or upper threads edited.

5 = Both replace and upper threads have edited this line and this line is ready to be written to the text file.

6 = The edited version of this line was written to the text file.

The int values in the "upper_replace_WorkDone" array are arranged as follows:

The "read_lines_from_file" method sets the line as 1 when it is read.

After the “ChangeToUpper” method finishes its operation, the current line value is increased by 2.

After the "Replace Space" method finishes its operation, its current line value is increased by 2.

The "WriteThreadJob" method, after writing the given line to the text file, edits it as 6 and that means all the necessary operations for that line are completed.

Mutexes :

We use a total of 6 mutexes.

Names and uses of mutexes:

“writeMutex” : This mutex is used between read threads. It ensures that the threads take the line they will read in order and prevents the same line from being read more than once. After a thread gets the index information of the line it will read, it releases the mutex with the "unlock" command and another thread can get the information of the line it will read.

“upperReplaceMutex” : This mutex is used to prevent upper and replace threads from entering the "UpperandReplaceGetWorkNumber" method at the same time. Thanks to this mutex, there is no conflict between the upper or replace threads and they can get the correct number of lines to run.

“writeToFileMutex” : It allows Write Threads to get the information of which line to write to the text file in order. The write thread, which receives the sequence number, releases the mutex with the "unlock" command after its operation is finished. If 2 different threads try to open a file at the same time, we may encounter a "Segmentation Fault" error. To prevent this, we release the mutex after closing the file.

“printTerminalMutex” : When too many threads use the "printf" function at the same time, we may encounter problems such as nested outputs in the terminal. To prevent this, each tread uses the printf function in turn, thanks to the “printTerminalMutex”. A thread "locks" the mutex before calling the printf function and "unlocks" it after using the printf function.

“readArrayMutex” : In LINUX operating systems, when different threads try to read the data in an array at the same time, it gives a "Segmentation Fault" error and to prevent this, we use this mutex while reading the "writeLineArray" array, so that different threads do not try to get information in the array at the same time.

“readDoneArrayMutex” : As said for the "readArrayMutex" mutex, this time we use this mutex to prevent reading the "upper_replace_WorkDone" array at the same time.