

**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING**

CSE 3033

Operating Systems

Project 3

Date: 11.01.2023

|  |  |  |
| --- | --- | --- |
|  | Student Id | Name Surname |
| 1 | 150119739 | Emir Ege EREN |
| 2 | 150119743 | Burak DURSUN |
| 3 | 150119745 | Haydar Taha TUNÇ |

**Introduction**

The purpose of the program is to read from a specified text file, modify its contents by converting the text to uppercase letters, replacing spaces with underscores, and then write the modified contents back to the same file. This is done using four different types of threads: Read Threads, Upper Threads, Replace Threads, and Write Threads. The program utilizes synchronization techniques such as semaphores and shared data, to ensure that the threads can access shared resources in a safe and consistent manner. The program will be executed with the user specifying the number of threads to be created for each type and the name of the file to be read from. The main thread creates all the other threads and waits for them to complete before exiting. The threads will also print information about their job on the screen and check if there are more jobs to do before exiting.

**Main part:**

1. Initializes and checks the command-line arguments provided by the user.
2. Reads the number of lines in the file and the line size information.
3. Initializes the shared data structures and semaphores.
4. Parses the command-line arguments and creates the threads using **pthread\_create**.
5. Waits for all the other threads to complete using **pthread\_join**.
6. At the end, the main thread exits after all the work has been done.

**Read part:**

1. The **read\_thread** function first acquires the semaphore **semRead**, which is used to protect the shared resources and prevent race conditions.
2. The function then calls the **getNextReadLine()** method, which returns the next line that needs to be read. The function continues reading lines until all lines have been read.
3. Once the function gets the next line number it needs to read, it calls the **read\_line()** method. This method reads the specified line from the file using a file descriptor and returns the line as a string.
4. The **read\_thread** function stores the line it read into the shared array by using the line number it got from the **getNextReadLine()** method.
5. The function then releases the semaphore **semRead** to allow other read threads to access the shared resources.
6. Lastly, the function exits and lets the main thread know that it has completed its task by returning NULL.

**Upper part:**

1. The **upper\_thread** function first acquires the semaphore **semUpper**, which is used to protect the shared resources and prevent race conditions.
2. The function then calls the **getNextUpperLine()** method, which returns the next line that needs to be converted to uppercase. The function continues to convert lines until all lines have been modified.
3. Once the function gets the next line number it needs to convert, it reads the line from the shared array.
4. It then iterates through the line, character by character, and using the **toUpper** method of **ctype** library, converts it to uppercase.
5. The **upper\_thread** function updates the shared array with the modified line by using the line number it got from the **getNextUpperLine()** method.
6. The function then releases the semaphore **semUpper** to allow other upper threads to access the shared resources.
7. Lastly, the function exits and lets the main thread know that it has completed its task by returning NULL.

**Replace part:**

1. The **replace\_thread** function first acquires the semaphore **semReplace**, which is used to protect the shared resources and prevent race conditions.
2. The function then calls the **getNextReplaceLine()** method, which returns the next line that needs to be replaced. The function continues to replace lines until all lines have been modified.
3. Once the function gets the next line number it needs to replace, it reads the line from the shared array.
4. It then iterates through the line, character by character, and checks if the character is a space. If it is, the function replaces it with an underscore character.
5. The **replace\_thread** function updates the shared array with the modified line by using the line number it got from the **getNextReplaceLine()** method.
6. The function then releases the semaphore **semReplace** to allow other replace threads to access the shared resources.
7. Lastly, the function exits and lets the main thread know that it has completed its task by returning NULL.

**Write part:**

1. The **write\_thread** function first acquires the semaphore **semWrite**, which is used to protect the shared resources and prevent race conditions.
2. The function then calls the **getNextWriteLine()** method, which returns the next line that needs to be written back to the file. The function continues to write lines back to the file until all lines have been modified.
3. Once the function gets the next line number it needs to write, it reads the modified line from the shared array
4. It writes the line back to the file by using the write system call, by specifying the file descriptor and the line offset.
5. The **write\_thread** function updates the shared array with the written status by using the line number it got from the **getNextWriteLine()** method.
6. The function then releases the semaphore **semWrite** to allow other write threads to access the shared resources.
7. Lastly, the function exits and lets the main thread know that it has completed its task by returning NULL.

**Screenshot:**





 metin içeren bir resim

Açıklama otomatik olarak oluşturuldu