Report on exercise #1

Matteo Corain S256654

Laboratory #12 – System and device programming – A.Y. 2018-19

The proposed solution for the exercise makes use of three types of data structures:

- FOLDER: it is used to store information on a directory, including its path (`TCHAR path[]`) and the associated variables for synchronization purposes (the `mutex` event, the `csReaders` critical section and the `nReaders` integer);
- RECORD: it is used to represent a single record in the servers' log files, including the IP of the user, his name, the date and time of access and the length of the operations;
- SHARED_VARS: it is used to hold data that is common to all threads, including the number of folders to analyze (`DWORD numberOfFolders`) and a dynamically allocated array of FOLDER structures (`LPFOLDER folders`).

The main thread, after having checked the number of parameters, opens the server list file, checking the correctness of the operation. First, the number of servers is read from the file and stored in the appropriate field of a SHARED_VARS structure; this is then used to allocate the array of folders. For each folder, the path is read from the file to memory and the synchronization variables are initialized; after this operation, the server list file is closed.

The main thread proceeds to parse the number of threads from the second command line argument and uses it to allocate an array of handles, which will be used to store references to the threads it is about to create. Immediately after, threads are created by making them run the `ThreadFunction` function and passing to them the SHARED_VARS structure filled before. Threads are finally joined, the synchronization variables are destroyed and the allocated dynamic memory is freed.

The function executed by the different threads initializes the random seed to its identifier, then indefinitely loops through the following operations:

- It generates three integer numbers, `n1`, `n2` and `n3`, in the interval 0-99;
- It sleeps for `n1 * 1000` milliseconds;
- It copies a pointer to the FOLDER structure at the `n3 * sv->numberOfFolders / 100` index of the folders array into a local variable;
- It decides, by comparing the value of `n2` against 50, whether it has to work as a reader (running the `Reader()` function) or a writer (running the `Writer()` function); in both cases, the pointer to the FOLDER structure to use is passed to the function.

The `Reader()` and `Writer()` functions implement the reader and writer synchronization problem, in the solution with priority to the readers, on the synchronization variables created for each single directory listed in the file parsed by the main thread.

Specifically, the `Reader()` function has an ingress section in which the thread, in mutual exclusion with other readers (achieved by protecting this part via the `csReaders` critical section), increments the number of current readers of the folder and, if no other reader is currently active, waits on the folder event until a writer has finished its operations; successive readers will not have to perform this operation and will pass directly to the central section of the function. The opposite tasks are performed in the egress section of the function, in which the counter of readers of the folder is decremented and, if no other reader is active, the folder event is signaled. From the writers' side, the protocol is much simpler; the `Writer()` function simply waits for the

folder event to be signaled and releases it at the end of the operations (writers must work in mutual exclusion both with readers and with other writers).

The central part of the two functions is very similar: in both cases, in fact, the function has to traverse the file system folder given as argument. In order to do so, the two functions open a search handle via `FindFirstFile()`, using as search path the path of the folder followed by a * wildcard. Then, protected by a termination handler, both of the functions enter in a do-while loop whose condition is given by the result of the `FindNextFile()` system call in order to list all the files in the target directory; finally, the search handle is closed via `FindClose()` before starting to execute the egress section of the function.

For each listed file, the `Reader()` function opens the file, checking the correctness of the operation, and reads all the file in fixed-size blocks, storing the read data in a RECORD structure. This is then used to update the total connection time in seconds (retrieved by means of the provided `GetSeconds()` function) and the last access time for the server (dates are compared by means of the provided `CompareDates()` function). The file is read in a strictly sequential fashion, so there is no need for using the OVERLAPPED structure.

The `Writer()` function again opens each listed file, checking the correctness of the operation, and reads records in strict sequence. For each record, however, different operations are performed:

- The `length` field is randomized by printing there a formatted string composed by a random number between 0 and 99 (connection hours) and two random numbers between 0 and 59 (connection minutes and seconds); all three numbers are printed on two digits (zero-prefixed, if necessary) and separated by colons;
- The `datetime` field is randomized by printing there a formatted string composed by a random number between 2000 and 2019 (for representing the year in a credible way), a random number between 1 and 12 (for representing the month) and a random number between 1 and the maximum number of days for the selected month, stored in the global array `daysPerMonth`, plus three random numbers representing the access time in terms of hours (0-23), minutes (0-59) and seconds (0-59); the string is formatted according to format “`%04d/%02d/%02d:%02d:%02d:%02d`”.

The updated line is then written back to file by means of a `WriteFile()` operation. Since after having read the line the file pointer has advanced, in this case it is necessary to make use of an OVERLAPPED structure to write the data in the correct position; after each write operation, the overlapped structure is updated by basically adding to the offset the size of the RECORD structure.