Report on exercise #2

Matteo Corain S256654

Laboratory #10 – System and device programming – A.Y. 2018-19

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

- SHARED_VARS: it is used to hold data that is common to all threads, including:
  - The synchronization variables: a HANDLE for storing a semaphore `semFileFound` and a vector of handles (LPHANDLE) called `eventsFileOk`;
  - The shared variables for inter-thread communication: a vector of strings (LPTSTR) called `filenames`, a vector of booleans (LPBOOL) called `finished` and an additional boolean (BOOL) called `kill`.
- VISIT_THREAD_DATA: it is passed as a data structure to a visiting thread, and it holds the index of the thread (`DWORD id`), the path that the thread has to visit (`LPCTSTR path`) and a pointer to the SHARED_VARS structure;
- COMPARE_THREAD_DATA: it is passed as a data structure to the comparing thread, and it holds the number of active threads (`DWORD nThreads`), their handles (`LPHANDLE threads`) and a pointer to the SHARED_VARS structure.

The main thread, after having checked the number of parameters, dynamically allocates:

- A vector (`threads`), with size `nThreads`, to store the handles related to the different threads;
- A vector (`vtd`), with size `nThreads-1`, of VISIT_THREAD_DATA to be passed to visiting threads;
- A shared vector (`sv.filenames`), with size `nThreads-1`, of string pointers in which the visiting threads will emit the discovered filename at each iteration;
- A shared vector (`sv.finished`), with size `nThreads-1`, of boolean values which the visiting threads will rise when they will finish their work;
- A shared vector (`sv.eventsFileOk`), with size `nThreads-1`, of handles to store the events on which the visiting threads will wait.

Then, it proceeds to create the `semFileFound` semaphore, on which the comparing thread will wait. After that, it proceeds to create `nThreads-1` visiting threads, passing to each of them an increasing integer, one of the arguments of the command line and a pointer to the local shared variables; the single comparing thread is also created, passing it the number of threads, the vector of handles and a pointer to the shared variables. The main finally proceeds to wait for all threads to complete, destroy the semaphore and free the allocated dynamic memory.

Each visiting thread starts by allocating memory to store three strings: `cwd`, `absolutePath` and the appropriate location in the shared `filenames` vector; then, it creates the event on which it will wait on (auto-reset, not signaled). After that, it unwraps the path received if it is relative using the information on the current working directory obtained via `GetCurrentDirectory()`; since we are working with threads, in fact, we cannot use the technique for changing the directory while exploring the file system, and we are needed to use absolute paths.

Successively, the thread will start the visit of the file system by calling `VisitPath()`; this function opens a search handle into the given path via `FindFirstFile()`, then use it to list all the contents of the directory via subsequent calls to `FindNextFile()`, taking care of not analyzing the current and its parent folder. For each entry, the thread writes on its location of the `filenames` shared variable (without needing any

synchronization, since each thread accesses a different location of the vector) the relative path of the discovered entry; then, it signals on the `semFileFound` semaphore and starts waiting on the appropriate event stored in the shared vector `eventsFileOk`. Once the comparing thread will unblock them, each visiting thread will check the shared variable `kill` in order to understand if the comparison has been unsuccessful and therefore the thread needs to terminate; in that case, the function returns false. Otherwise, it will check if the entry refers to a directory and, in case, perform a recursive call after having produced the new absolute and relative paths to visit. Finally, the function closes the search handle with `FindClose()` and returns to the thread function.

The thread function frees the allocated strings, destroys its associated event and proceeds to set its termination flag in the shared `finished` variable; finally, it signals on the `semFileFound` semaphore to unlock the comparing thread.

The comparing thread loops until the global `kill` variable is false, performing the following actions:

- It waits for `nThread-1` times on the `semFileFound` semaphore, in order to let all the visiting threads producing a filename (or terminating);
- It checks whether all threads have terminated at once; if that happens, it prints a message stating that the two given trees are equal and sets the `kill` flag;
- It checks whether at least one thread has terminated; if that happens, it prints a message stating that the two given trees are only partially equal and sets the `kill` flag;
- It checks whether all the filenames discovered by all the visiting threads are equal; if that does not happen, it prints a message stating that the two trees are not equal and sets the `kill` flag;
- If none of the previous situation happened (`kill` is still false), it prints a message stating that the discovered entries are equal; then, in any case, it sets all the events related to the different visiting threads in order to unblock them.

To perform those actions, two functions have been introduced:

- `CheckTermination()`: it runs a loop checking the termination flags of the different threads; if the boolean parameter `all` is true, the function returns true only if all the threads have terminated (first check of the comparing thread), otherwise it will return true if at least one has terminated (second check of the comparing thread);
- `CheckFilenames()`: it runs a loop to check, in pairs, the strings produced by the different visiting threads, returning false if at least a difference is found.