Report on exercise #3C

Matteo Corain S256654

Laboratory #9 – 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 CRITICAL_SECTION called csPrint and two HANDLEs, used for storing two events, called eventStartPrint and eventStartEnd;
  - The shared variables for inter-thread communication: a BOOL called finished, a DWORD called currentTID and a vector of TCHARs (of length LINELEN, equal to 4096) called producedLine.
- VISIT_THREAD_DATA: it is passed as a data structure to a visiting thread, and it holds the path that the thread has to visit (LPCTSTR path) and a pointer to the SHARED_VARS structure;
- PRINT_THREAD_DATA: it is passed as a data structure to the printing thread, and it holds the number of active threads (DWORD nThreads), their identifiers (LPDWORD threadIds) and a pointer to the SHARED_VARS structure.

The main thread, after having checked the number of parameters, allocates three vectors to store respectively the handles related to the different threads to be created, the identifiers of the threads to be created and the VISIT_THREAD_DATA structures to be passed to visiting threads. Then, it proceeds to create the synchronization variables inside its local SHARED_VARS structure (a critical section and two events are used for the purpose of the exercise). After that, it proceeds to create nThreads-1 visiting threads, passing to each of them one of the arguments of the command line and a pointer to the local shared variables; the single printing thread is also created, passing it the number of threads, the vector of identifiers and a pointer to the shared variables. The main finally proceeds to wait for all threads to complete, destroy the synchronization objects and free the allocated dynamic memory.

Each visiting thread first of all unwraps the path received if it is relative; 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. Then, it 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, a line is emitted via EmitLine(), containing a number of spaces for indentation, the current thread identifier and the name of the directory or file; if the entry refers to a directory, a recursive call is performed. Finally, the function closes the search handle with FindClose() and returns to the thread function, which calls SignalTermination() and terminates.

The two functions EmitLine() and SignalTermination() are used to manage the interaction between the visiting thread and the printing thread. Specifically, they acquire the critical section in order to work in mutual exclusion among all visiting threads; then, they access the shared variables, setting them accordingly to their specific function:

- finished is set to TRUE by SignalTermination() and to FALSE by EmitLine();
- currentTID is set to the current thread identifier;
- producedLine is set by EmitLine() according to the given format, by means of a call to the library function _vstprintf().

After having set the shared variables, the two functions set the `eventStartPrint` to wake up the printing thread and wait for it to complete by setting the `eventEndPrint`, then they leave the critical section.

The printing thread allocates a vector of file pointers, associating to each visiting thread a file named after its thread identifier, opened in write mode. After that, it starts a loop which terminates when its local variable `finishedThreads` reaches the number of visiting threads; inside the loop, the thread waits for the `eventStartPrint` to be raised by a visiting thread. When the thread is unlocked, it linearly scans the array of file pointers until it finds the one related to the thread that performed the unlock (its identifier is stored in the `currentTID` shared variable); then, it performs the following actions:

- If `finished` is set (visiting thread has called `SignalTermination()`), it closes its file pointer, prints the results of the visit by means of the `PrintResults()` function and increments the number of finished threads;
- If `finished` is not set (visiting thread has called `EmitLine()`), it proceeds to output the emitted line on the file pointer related to the visiting thread.

The `PrintResults()` function simply opens a new pointer to the file related to a thread for which the identifier is given, and prints it line by line on the standard output. After all visiting threads have finished, the printing thread frees the array of file pointers (all should be properly closed at this point) and returns.