**Topic:** Synchronization

**Resources:** The program **Race_Condition** creates two threads which repeatedly execute a critical section accessing global variables. The critical sections are protected by mutexes, but the synchronization is not correct.

**Remark:** For grading the assignment, Ex. 1c, 1d, and - above all - the design and implementation of the program in Ex. 3 as well as the explanation of the synchronization used in the program, and the design and documentation of the test scenario and results will be most important. The documentation of Ex. 3 should be understandable without having to read the source code. However, include the code as an appendix nevertheless.

**Exercise 1**

Compile and run the program **Race_Condition**. Normally, it will report a race condition within a few seconds (on an SMP system), but sometimes it may take longer (races are unpredictable!).

**a)** Study the source code **Race_Condition.cpp**. Pay attention to
- how threads are created,
- how the primary thread synchronizes with the termination of the other threads,
- how a mutex is used to achieve mutual exclusion,
- the use of local and global variables, and the attribute volatile.
(No documentation)

**b)** Deduce from the source code, and in particular from the condition used for detecting a race condition, what the program is supposed to do.

**c)** There are several ways in which race conditions can occur in this program. Describe at least two significantly different possibilities of how a race condition can occur that will be detected by the program.

**d)** The program does not detect all occurrences of a race condition. Give an example of how a race condition can remain undetected (there are several ways how this can happen).

**e)** What is the purpose of the local variable **saved_sum**? Does the use of **saved_sum** always work as intended?

**f)** Correct the synchronization in the program. Do not change the program logic (e.g. by replacing the critical section by `sum=sum+2`, though this would still be critical).

If you want to test your new synchronization, run your program again, and let it run for a long time. This may (or may not) show any remaining mistakes in the synchronization, but can of course not prove its correctness.
**Exercise 2**

a) Study at least one of the solutions implementing a reader-writer synchronization which are available on Moodle:
   - The solution taken from the Tanenbaum book using only mutexes.
   - The solution taken from the MSDN using auto-reset and manual-reset events.
   Pay attention to how the basic requirements (shared read, exclusive write) are achieved. (No documentation)

b) Using the code of the solution you studied in a), analyze and explain what happens in the following situation:
   Assume that a writer requests access while readers are active. Will new read requests that come in before the write access can be granted be blocked or granted? (The two given solutions behave differently in this situation!)

c) Read about the slim reader/writer locks introduced with Windows Vista/Server 2008 in the MSDN (or some other reliable source). (No documentation)

**Exercise 3**

Write a program exhibiting the reader-writer semantic with several reader threads and (at least) one writer thread.

For the synchronization, you may want to use one of the following mechanisms:
   - The Tanenbaum solution, re-written for a Windows system (take the source code of the program Race_Condition from exercise 1 as a guideline on how to use the needed system calls).
   - The solution with event objects taken from the MSDN.
   - The slim reader/writer locks (on a Vista or W7/W8/W10 system).

However, you may also work on any other operating system, and/or use any other synchronization mechanism. If you do so, describe and explain the mechanism as well as its implementation and use in as much detail as is necessary to fully understand your program.

Find a way to test the correctness of your program.
One possibility: The reader and writer threads output messages like:
- Reader <n> requesting read access.
- Reader <n> was denied access, blocking. (Request access with timeout 0, output the message, and then re-request access in blocking mode.)
- Reader <n> was granted read access.
- Reader <n> finished reading.
- Similar messages for the writer.

Another idea would be to output the number of current readers together with the current state of the writer (no intention to write – requested write access – writing). Whatever you do, you will most probably need additional synchronization for the output!

Let the readers and writer(s) run in an infinite loop with random waits in between their access requests, and then search the output for occurrences of
- the situations showing the required behavior of any reader-writer lock
  (3 situations, resp. 4 when you have several writers)
- the situation described in 2b).

Identify and document only those. If you chose to use Windows’ slim reader/ writer locks, also test and document the relevant behavior to clarify the mystical sentence in the MSDN: “There is no guarantee about the order in which threads that request ownership will be granted ownership: SRW locks are neither fair nor FIFO.”