System and Device Programming

Examination Test – Programming Part
11 September 2017

Examination Time: 1h 45min. Evaluation. 18 marks.
Textbooks and/or course material allowed.

The standard sum-and-shift algorithm to multiply two numbers works as illustrated by the following example:

			0	3	2	x
			2	4	6	=
0	0	0	1	9	2	+
0	0	1	2	8		+
0	0	6	4			=
0	0	7	8	7	2	

During this process notice that:

- To avoid overflow, two factors of n digits generate a product of $(2 \cdot n)$ digits.
- All digits of the first factor are multiplied by all digits of the second factor starting from the least significant digit and moving to the most significant digit. In each product it is necessary to properly propagate the carry value along the product chain.
- All partial results are added together shifted of one position.

The candidate has to write a Windows 32 application to multiply very long integer numbers stored in as set of files with the following specifications.

The application receives an integer number n and a directory name on the command line. All files with extension "`.bin`" in the (one-level, i.e., flat) directory, stores records with three fields. The first two fields of each record, represent the two factors of n digits. Each digit is stored as a 32-bit integer value. The third field includes $(2 \cdot n)$ 32-bit integer values, initially all equal to zeros, and it represents the result.
For example, in the previous picture, $n = 3$, the two factors plus the result would be stored in binary form as a sequence of digits, where each digit will be represented as a 32-bit integer value:

0 3 2	2 4 6	0 0 0 0 0 0
First factor	Second factor	Result

The program has to compute the product for all records of all files, and it has to store the results in the last field of each record. Then, at the end of this computation the previous record will be:

0 3 2	2 4 6	0 0 7 8 7 2
First factor	Second factor	Result

To do that, the program has to run $n + 1$ threads:

- Threads $[0, n-1]$ are *product* threads. Within this set, thread i is in charge of multiplying the first factor for digit number i of the second factor. For the previous example, thread 0 would computed $032 \cdot 6$, thread 1 would compute $032 \cdot 4$, etc. All threads $i \in [0, n-1]$ have to work concurrently, but once they have computed one product, they have to wait thread number n before moving to the next product (stored in the next record of the same file or into the next file).
- Thread number $n + 1$ is the *addition* thread. It computes the sum off all partial products, and it stores the result in the last $(2 \cdot n)$ digits of the current record. The *addition* thread has to compute one sum as soon as one *product* thread has terminated. For the previous example, if we suppose that product threads terminate is the same order they are run, the sum thread would compute $000000 + 000192$, then $000192 + 001280$, etc. Once the result has been stored in the file, the *addition* thread has to wake-up all *product* threads such that they can restart the process.

Notice that it is requested an implementation where the activation of the *addition* thread is extremely efficient, and this cannot be obtained with function `WaitForMultipleObject` as n is usually much larger than 64.