Real Time Embedded Systems
Producer-Consumer Problem (POSIX threads)

- Papadakis Charalampos AEM:9128 e-mail:papadakic@ece.auth.gr

- Source code can be found in the link: https://www.dropbox.com/s/aeml3jvl8c6ogqx/1stAssignment.c?dl=0

Problem and Functions Description:

In this assignment we are asked to modify the classic Producer-Consumer problem using Pthreads programming so that it is working for multiple producers and multiple consumers. In more details, each producer is adding in a queue multiple functions that have to be "done", so the consumers are responsible for the execution of this functions, while they ,also have to delete the completed functions from the queue.

- **void *producer (void *q):**

The producer function is taking as argument the FIFO queue(the queue has size defined to a large number (QUEUESIZE) in which the functions must be added. This function selects a random small function(work) out of the 5 possible and a random argument out of the 10 possible , so it adds in the queue a workFunction object (the workFunction structure contains of a small function and an argument). Each producer adds to the queue LOOP (large number) workFunction objects,and after that its job is completed.

- **void *consumer (void *q):**

The consumer function is ,also, taking as argument the FIFO queue in which are the functions that must be executed. This function uses an infinite loop (while 1) in which it is "choosing" a workFunction object from the queue ,deletes it , and then executes the function(work) of the object using the right argument (arg). After every producer is completed and the queue is empty ,all the consumers are freed(pthread_cond_broadcast) the loop breaks and the consumers job is done.

- **int main():**

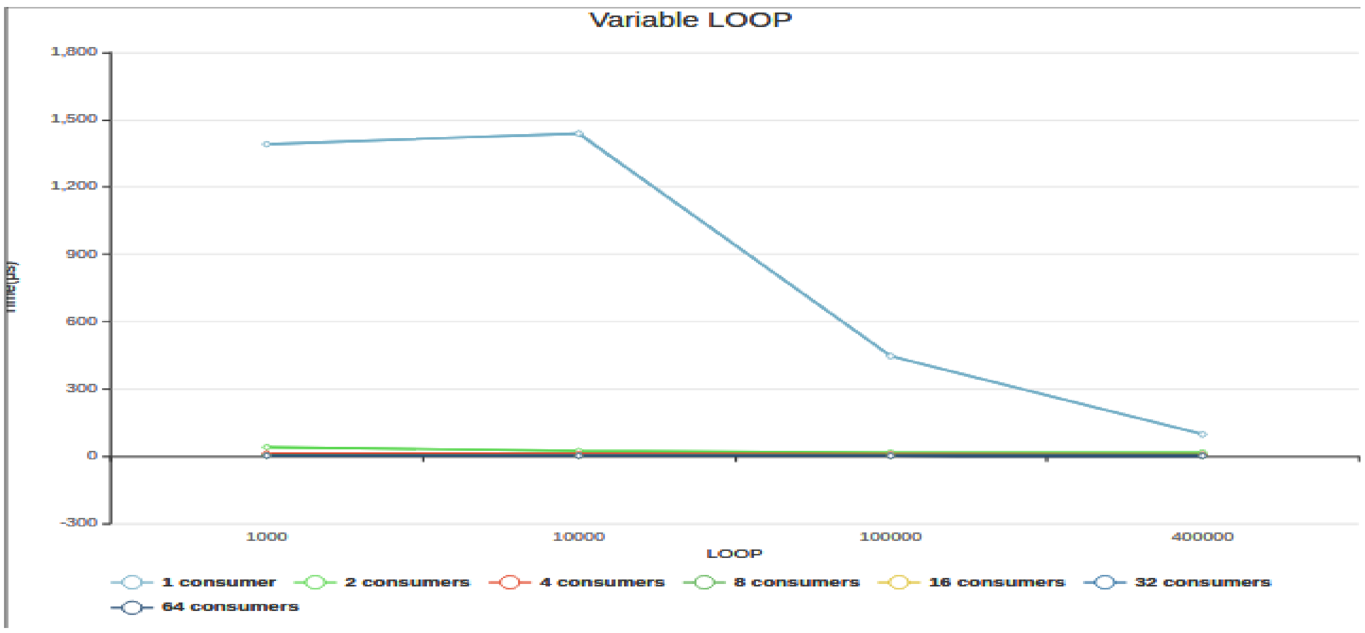
In the main function we create P threads for the producers and Q threads for the consumers. After the creation we are using pthread_join for every consumer and producer thread.

- **work functions:**

In the program there are 5 possible functions that a producer can select from to add to the queue. The functions are printSinX, printCosX, squareX, squareRootX, logX.

Time results:

I have chosen a QUEUESIZE 500 after experimenting with various values and realizing that further increasing of the size would be useless, as the threads of my Laptop would never lead to the queue being full,and most of the times it would be half empty or so. Firstly ,I ran with various LOOP values. Because of the little space in this report, I will present the results in a diagram,while running the program with 2 Producers , QUEUESIZE set to 500, and changing LOOP values.



As I can see there are not many differences, however when the LOOP is getting bigger the systems seems to become more stable and more efficient.
Below, the time results are showing the average waiting time that took to an object from the time that it was submitted to the queue by a producer,until the time it was received by a consumer. **However, in the results below ,the time that took for the consumer to run the work function is not included**. During the calculation I set the LOOP to 400,000 and the QUEUESIZE to 500.

All the results of each combination in the next table are recorded in μs:

		Consumers						
		1	2	4	8	16	32	64
Producers	**1**	398.660	173.333	1.8208	1.2884	1.7159	1.1416	**1.1179**
	2	586.217	189.425	7.5487	1.9674	1.6989	1.6273	1.5498
	4	1568.435	434.130	77.873	20.358	2.326	2.1579	2.1726
	8	1648.959	1542.495	496.767	33.777	2.3228	2.3985	2.4412
	16	2284.231	1892.982	1394.363	1445.074	88.848	2.6761	2.6397

<u>Conclusion-Observations:</u>

- It is shown above that the smallest average waiting time was provided by the combination 1 produver and 64 consumers.
- I have observed that as the producers and consumers increase (more workFunction objects), the system becomes more stable, maybe because the queue is fuller now.
- It is normal that when we have more producers than consumers the time rises in really big values, and that is because the objects that are submitted are too many for the amount of consumers we have to handle them.
- The results occurred after multiple executions of the program for every producer-consumer combination, using average values.
- After 128 consumers I saw no time improvement, so I stopped at 64.
- More details on the functions will be found in comments on the source code.
- The time results were recorded in a laptop with Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz and 4 processors.