Programming assignment 1
ECE/CS 5510

{ayushchatur,aakashc} @vt.edu

github: https://github.com/ayushchatur/mpp_prog1

House Keeping Notes :

The Java version on rlogin server is 11. This project needs to be compiled with java 8 per instructions in syllabus and elsewhere communicated. Thus, we do not advise compiling and running on rlogin server. The project should be compiled with a system having java version 8 or equivalent and then the jar then needs to be copied and run per (command line) instructions below.

The programming assignments can either be run via netbeans ide or command line. Please follow the respective **instructions.md** file in github repository

We have migrated Test files to src/mutex package in order to remove dependencies on junit and make it more convenient to run via command line.

For instructions on running via command line follow **cmd.md**
The data in table below is run using taskset.sh file provided

Lock Implementation:

For the implementation of algorithms we have to use AtomicInterger and AtomicBoolean Java variables this is due to the fact the java does not guarantee sequential consistency or linearizability[1]. Therefore, using the atomic constructs we make sure there is no instruction reordering or optimization done via JIT compiler of JAVA runtime. The same behaviour can also be achieved via using volatile keyword in java. As a support for our claim we have also implemented the Bakery algorithm without using Atomic variables, This can be run using choice of algorithm as 4 (refer cmd.md file in package). Despite being the correct implementation all the threads enter the wait when there are more than 2 threads used, this shows the necessity of atomic constructs when using JAVA.

Correctness Evaluation :

Test - To check the validity of the algorithm of lock we use a simple counter increment, each thread increments the counter by a value of 1 each time it enters the critical section. And each thread does this a determined number of times. Finally we match the counter value with the anticipated final value to validate the correctness.

- *Bakery Lock*

Timings – The timings for Bakery Lock implementation are tabulated in Table I given below

Table I

Bakery Lock	**Averge time (nano sec)**	**No of iterations =5**														
Thread #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
# of total threads																
1	69															
2	120	123														
4	367	409	404	399												
6	254	199	182	185	134	146										
8	224	238	241	146	161	134	154	126								
10	144	191	181	181	203	223	183	210	283	167						
12	313	320	340	487	525	541	508	326	505	361	174	343				
14	179	181	262	211	222	235	193	185	189	183	205	295	278	197		
16	199	257	446	268	444	456	442	298	443	437	463	470	474	462	242	197

- *Filter Lock*

Timings – The timings for Bakery Lock implementation are tabulated in Table II given below

Table II

Filter lock	**Averge time(nano sec)**	**No of iterations =5**														
Thread #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
# of total threads																
1	62															
2	220	99														
4	150	153	274	180												
6	462	461	226	305	235	236										

8	501	522	463	428	381	312	386	346								
10	1103	1247	647	1137	1073	1495	996	1315	965	1090						
12	2222	2053	1875	2101	1903	1975	1911	2059	2030	1825	1998	1820				
14	2243	2364	2388	2417	2332	2359	2314	2240	2443	2231	2226	2421	2404	2303		
16	145	1196	1271	1181	1259	1148	1048	1185	1036	1179	1159	1111	1301	970	1099	1051

- *Re-entrant Lock*

  Timings – The timings for Bakery Lock implementation are tabulated in Table III given below
  Correctness evaluation -

Table III

Re-Entrant lock	**Average time(nano sec)**	**No of iterations =5**														
Thread #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
# of total threads																
1	96															
2	225	96														
4	81	98	74	69												
6	98	86	89	111	92	85										
8	83	87	90	86	83	93	92	83								
10	83	97	103	85	85	82	79	83	86	85						
12	116	143	121	116	108	103	114	116	115	111	116	118				
14	167	144	125	92	152	112	100	112	106	103	104	121	97	106		
16	98	105	133	95	89	89	97	83	89	99	89	87	86	101	108	126

References:

[1] The Art of Multiprocessor Programming. Maurice Herlihy, Nir Shavit, 2008, **Section 7.1**,