Distributed Algorithms CPSC-561
Assignment 1

Michael McCulloch

1 Funny Lock

Mutual exclusion is violated by two processes P and Q, by the execution order: $P_1, P_2, P_3, P_5, P_2, P_{CS}, P_7, Q_1, Q_2, Q_3, P_1, P_2, P_{CS}, Q_5, Q_2, Q_{CS}$

Deadlock-freedom is violated by two processes P and Q, by the execution order: P_1, Q_1, P_2, Q_2

2 Lousy Lock

Mutual Exclusion: Any process which enters the critical section must have it's flag with value two, and then must have checked that the other process's flag is not two. If two processes are in the critical section, then their flags must both be two, and the others may not have been two. A contradiction.

Deadlock Freedom: If > 2 processes are trying to enter the critical section, both will have $flag_i = 1$, Any process which then executes line 3 will fail, and will not set it's turn flag. Since `turn` may only hold one value, this excludes p_0 on the first invocation of the lock method, and in subsequent invocations, the last process to enter the critical section, in either case, these processes get a free pass around this loop, and will progress, satisfying deadlock freedom.

There is a special case where two processes may execute line 5 simultaneously, and so we must argue deadlock freedom for the loop at line 7 as well. The execution is as follows: $Q_1, Q_2, Q_{3.1}, P_1, P_2, P_4, Q_{3.2}, Q_2, Q_4, P_5, Q_5$. (With 3.1 being the `if` portion, and 3.2, the `then` portion of the conditional). On the second of this loop however, P will fail the test at line 3, and loop here.

Allowing Q to reach line 5 alone, and make progress. Satisfying deadlock Freedom.

Starvation Freedom is not satisfied here, as if a thread sleeps before executing line 3, another thread may re-acquire the lock infinitly many times.

3 Print Primes variants

For a range of 62.5 million to 125 million:

Threads	1	4	8	16	32	Sum
LLLock	87.62	24.28	20.21	33.77	56.37	222.25
OTLock	88.06	22.61	18.23	18.52	19.59	167.01
Atomic	87.69	22.4	18	18.02	17.99	164.1
Sum	263.37	69.29	56.44	70.31	93.95	

Vertical Interpretation The long lived lock appears to perform the most poorly. This is likely because once a thread has called it, it must play the filter game, whereas the one time lock returns immediately for threads that are too late, and they may move forward. The atomic lock uses hardware resources and naturally has an advantage.

Horizontal Interpretation For one thread, there is no competition, it rips through the lock() methods on it's own at about the same rate for each. When more than one thread is involved however, there is a somewhat significant speedup up until the number of thread matches the number of physical processor cores. My machine has 4 physical cores with hyperthreading, so context switching is still happening, but on a hardware optimized level instead. For 8 threads, I would expect to see a greater speedup on a true 8 core, non-hyper threaded machine. Once the number of threads exceeds the number of logical processors however, these threads begin competing more agressively for time on the scheduler, and so more time is spent on context switches. The exception to this is in the atomic locks, which is likely due to some optimization magic on either the hardware or JVM level.

Variant 2 This variant failed to complete because of significant memory allocation costs. A new OTLock must be allocated for each number, in this case, 62.5 million OTLocks were created, occupying at least 8GB of memory. This causes java to run out of heap space unless the JVM is allowed more.