Concurrent Algorithms and Data Structures (Sheet #1)

Marius Gavrilescu

1. We know the enqueue of 4 finishes before the enqueue of 6 starts, so the queue must have 4 before 6. As the enqueue of 5 is concurrent with both of the other operations, it can happen either before both of them, or between them, or after both of them. Therefore the queue's final state can be 4,5,6 or 5,4,6 or 4,6,5.

4. The doorway section is the first two lines of the `lock` method, and the waiting section is the third line (the while loop).

   Suppose A finishes its doorway before B begins its doorway, and B enters the critical section before A.

   At the point when B finishes its doorway, we know A has finished its doorway but has not entered the critical section, so it must be waiting. Then both flags are set to 1, and "victim" is set to B, because B's doorway section set "victim" last. But at this point in time "B" will be forced to wait, because "A"'s flag is up and "B" is the victim, and when "A" has an opportunity to run again it will see it is no longer the victim and therefore it can enter the critical section. This contradicts our assumption of B entering the critical section before A. So the lock is first-come-first-served.

5. The lock satisfies mutual exclusion. Suppose both A and B entered the critical section at the same time, and assume A entered first. Then at the time A finished the "lock" function we know "busy" is true (because it was false when A exited the inner loop, and A set it true) and "turn" is "A". Then there are two cases: B finished the inner loop or B did not finish the inner loop at the point that A finished the outer loop.

   If B did not finish the inner loop, then it will not finish it before "A" calls unlock because "busy" is true.

   If B finished the inner loop, then it will set "busy" to true (a no-op), see that "turn" is "A", and loop again, entering the inner loop. Now we are back to the previous case.

   So B cannot enter the critical section, thus satisfying mutual exclusion.

   The lock is not deadlock-free. It could happen in the example above that "B" is at the beginning of the inner loop (between lines 8 and 9) and "A" has just left the inner loop (so it is between lines 10 and 11). Now A runs, sets "busy" to true. Then B runs, sets "turn" to "B". Now B cannot leave the inner loop (because "busy" is true) and A cannot leave the outer loop (because "turn" is "B"). A will loop again and both threads will be stuck inside the inner loop.

   Since the lock is not deadlock-free, it cannot be starvation-free.

6. We can recursively prove that no two threads act as thread 0 or 1 for the same lock. Start from the bottom layer of internal nodes. Clearly those locks satisfy this property, because only the left leaf can act as thread 0 and only the right leaf can act as thread 1.

   Now suppose that locks on level `n` satisfy this property, and take any lock "L"on level `n-1`. For a thread to act as thread 0 on "L", it must hold the lock in the left child of "L". But that is a Peterson lock where only one thread can act as thread 0 and only one thread can act as thread 1 at the same time. So "L"'s left child satisfies mutual exclusion, therefore only one thread can act as thread 0 on "L" at the same time. The same argument can be used to show only one thread can act as thread 1 on "L" at the same time. So locks on level `n-1` also satisfy this property.

   Therefore all locks satisfy the property that only one thread can act as thread 0 and only one thread can act as thread 1 at the same time. This means all locks will "work correctly", meaning that they individually satisfy mutual exclusion and starvation freedom as was proven in the lecture notes.

   Since only two threads can act on the root lock at the same time (and one is thread 0, one is thread 1), then we maintain mutual exclusion for the entire tree.

   Now to prove starvation freedom take any thread and consider the chain of locks it needs to acquire in order to enter the critical section. Since each of these locks is individually starvation free, we know it will eventually succeed in acquiring the first lock it needs, then it will eventually succeed in acquiring the second lock it needs, and so on until it eventually succeeds in acquiring the root lock. So the system is starvation free.

7. The modified Bakery Algorithm does not satisfy mutual exclusion if we manage to spawn enough threads to cause the label to wrap around. Suppose we have only the thread with label $2^{31} - 1$ running inside the critical section. Now when a new thread comes, it gets label -2^{31}, it calls `canEnter` and this method sees that the only thread with its flag set is $2^{31} - 1$, which has a larger label than me. So this thread is able to enter the critical section even though the thread $2^{31} - 1$ is still inside the critical section, thus breaking mutual exclusion.

11. If our object satisfies the top definition then the bottom definition is trivially satisfied, because we only consider a subset of the histories (the complete ones) and for them `complete(H')=H`.

    If our object satisfies the bottom definition, then for every incomplete well-formed history we know that all calls will eventually return (because our object is deadlock-free), so we can extend the history with the necessary returns, and this history is complete and well-formed, so by applying the bottom definition it satisfies the properties.

12. A linearization of the sub-history would be:

    B gets the key, it is 3. A deletes it. D sets the key to 0. C sets the key to 2. E gets the key, it is 2.

    If we assume the full history can be linearized, F's operation starts after everything else has finished, so we need the sub-history above to finish with the value 0. There is only one thread that sets the value to 0 (namely D). But since that operation returns 0, we know the value was either missing or already 0. It cannot be already 0, because no other thread writes 0 to it. So it must be missing. This means the previous write must be the delete performed by thread A.

    We therefore see that D's operation must be the last write, and the second-to-last write must be A's delete. But we know that at some point after A's delete finishes, E must see a value of 2. This is impossible, because there is no write between A deleting and D writing 0. This is a contradiction, so the full history cannot be linearized.