COMP9154

Foundations of Concurrency

I declare that all of the work submitted for this exam is my own work, completed without assistance from anyone else.

Part I

Question 1 (10 marks)

Answer the following, briefly and in your own words.

(a) (2 marks) What is a linear-time property?

The linear time property specifies the subset of behaviours that should only occur for the program.

(b) (2 marks) Why does Owicki-Gries require interference freedom checks?

Because even the correctness of each program running in parallel is proved, the state of one program may also be falsified by others if they are not interference free.

(c) (2 marks) How do distributed consensus algorithms get around the FLP theorem?

They try to devise a message delivering system with synchronous or partially synchronous protocol achieved by for example: time-out mechanism, message communicating between nodes. In a message communicating based algorithm, they reach an agreement by listening to others and voting on the majority of correct nodes.

(d) (2 marks) Why does the King algorithm require one more round than the number of traitors?

Because this ensures that there is at least one king that is not traitor. With the non-traitor king's message, the generals can reach to an agreement.

(e) (2 marks) What is the difference between a permission-based and a token-based distributed mutual exclusion algorithm?

Permission-based algorithm is implemented based on the communication of one node over all other nodes, once it gets the permission of others, it can proceed to the critical section. However, token-based algorithm only requires a node to receive one special permission, in other word, receive the token to enter the critical section.

Question 2 (15 marks)

(a) (5 marks) Of the four main critical section desiderata, which ones are satisfied by this algorithm? For any properties that are not satisfied, describe a behaviour that is in violation.

Eventual entry is violated in this algorithm.

One of the cases is that these programs repeat the bolded sequence forever thus q can never enter the critical section: $p_1\ p_2\ p_3\ p_4\ q_1\ q_2\ q_3\ q_4\ \ \boldsymbol{q_5\ p_5\ q_6\ p_1\ p_2\ q_3\ q_4\ p_3\ p_4}\ \ q_5\ p_5 \cdots$

In this case, p enters the cs and set x false firstly, and then q passed q_5 for waiting x to be false, and then p finishes the non-cs rapidly and set x true before q proceeds to q_3.

(b) (5 marks) Suppose we rewrite process p to be exactly like process q, but with the roles of x and y swapped. Would this change your answer to the previous question?

Eventual entry and **mutual exclusion** are not satisfied.

Assume that this time, process p also consists of 8 lines, from p_1 to p_8.

For eventual entry, the behaviour is similar to what we have in question a), p rapidly pick up the permission to enter cs before q moves on indefinitely which is a live lock.

For mutual exclusion, there is a case that p and q run interleaved line by line in this pattern:

$$p_n q_n\ \ p_{(n \bmod 8)+1} q_{(n \bmod 8)+1} \cdots$$

When they both arrive line 4 and set x and y false, and then move to line 3. Both of them will be permitted to jump to line 7, which is the critical section.

(c) (5 marks) Why are there no algorithms with only a single bit of shared state? You don't have to produce a formal proof, but try to make a convincing informal argument. For the purposes of this question, we consider an algorithm correct if it satisfies at least mutual exclusion and deadlock freedom.

There are two potential algorithms for a single bit of shared state. One is to make the bit as a "lock" and the other is to treat the bit as "turn".

P	Q
$lock = 0$	
$non-cs$	$non-cs$
$wait\ \neg lock$	$wait\ \neg lock$
$lock = 1$	$lock = 1$
cs	cs
$lock = 0$	$lock = 0$

P	Q
$turn = 0$	
$non-cs$	$non-cs$
$wait\ \neg turn$	$wait\ turn$
cs	cs
$turn = 1$	$turn = 0$

With lock, p and q can pick up the lock together if they execute interleaved line by line. As a result, they will enter critical section at the same time and violate the mutual exclusion requirement.

With turn, it is possible that one of them stuck in non-critical section and another one is trying to enter the critical section. For example, p is waiting for turn to be 0 however q is trapped in the non-cs forever then a dead lock occurs.

Question 3 (15 marks)

Assume you have an underlying monitor implementation I with priority ordering E < S < W. Suppose we would like a monitor M to behave as if it had priority ordering E < W < S. Show how to implement our desired monitor M using our underlying implementation I.

The pseudo code for M is:

```
can_enter = true;
start = false;
signalling = false;
execute_wait = false

enter_M()
    can_enter;        //Used for blocking E
    can_enter = false;
    enter_I();

leave_M()
    if !execute_wait:
        leave_I();    //Avoid double leaving
        execute_wait = false;
    start = true;     //Awake W if it's called by S
    can_enter = ¬signalling; //E can proceed when S and W finish

WaitC_M(cond)
    WaitC_I(cond);
    leave_I();        //Let S to start before W.
    start;            //Waiting for S to finish
    execute_wait = true;
    signalling = false; // E can proceed

SignalC_M(cond)
    signalling = true;//E cannot start between S and W.
    start = false;    //Prevent W from starting before S
    SignalC_I(cond);  //Continue after leave_I()
```

This code snippets ensures that W is always executed after an S by *start*. And also, E is executed only when neither S nor W is executing *can_enter and signalling*. Also, *execite_wait* is used to avoid calling leave_I() two times by W.

The assumption is that leave_M() will be called after a task leaves the monitor.

Question 4 (10 marks)

(a) (5 marks) Consider the LTL formula $\diamond \blacksquare \diamond \varphi$. Give a simpler but logically equivalent formula. Explain why it's equivalent.

The equivalent formula is:

$$\square \diamond \varphi$$

$\square \diamond \varphi$ can be descripted as:

For all $i \geq 0$, there exists a $j \geq i$ so that φ happens after jth. (1)

$\diamond \square \diamond \varphi$ can be descripted as:

There exists a $k \geq 0$, so that for all $i \geq k$,

there exists a $j \geq i$ so that φ happens after jth behaviour. (2)

It is straight forward that $\square \diamond \varphi \Rightarrow \diamond \square \diamond \varphi$. For the other side, we can prove it by induction:

Let's imagine a series of states:

1. Assume that in the base case that there exists $k = 0$, so that for all $i \geq 0$, there exists a $j \geq i$ so that φ happens.
2. Then we insert new states to the start of the series. Assume that when we have inserted n nodes, there exists $k = n$, so that for all $i \geq n$, there exists a $j \geq i + n$ so that φ happens.
3. If we insert one more state to the beginning, then for all $i \geq n$ there is always an $j \geq i + n + 1$ so that φ happens.
4. Starting from $n = 0$, we can deduce that for all $i \geq 0$, there is always an $j \geq i$ so that φ happens. Any series can be built by inserting elements to the beginning from the base case.

Thus, we have proved that $\diamond \square \diamond \varphi \Rightarrow \square \diamond \varphi$ by induction.

As a result, we have $\diamond \square \diamond \varphi \Leftrightarrow \square \diamond \varphi$.

(b) (5 marks) Let the CCS process P be defined as follows: $\boldsymbol{P} = (\boldsymbol{x}.\boldsymbol{P})\backslash \boldsymbol{x}$ Simplify the following CCS expression step by step: $(\boldsymbol{x}.(\boldsymbol{P}\backslash \boldsymbol{x}))\backslash \boldsymbol{x}$

$$\begin{aligned}
&\big(x.(P\backslash x)\big)\backslash x \\
&= \Big(x.\big((x.P)\backslash x\backslash x\big)\Big)\backslash x \\
&= \Big(x.\big((x.P)\backslash x\big)\Big)\backslash x \\
&= (x.P)\backslash x \\
&= P
\end{aligned}$$

Part II

Question 5 (15 marks)

(a) (5 marks) Section 1 of the paper includes an extensive summary of the state of the art in 1996. How did Michael and Scott's paper improve the state of the art?

It offers a solution that is mostly free of ABA problems implemented with only *compare_and_swap* while ensuring the non-blocking property. Also, the performance of their solution outperformed the state of the art. In addition, it ensures the safety assumption.

(b) (3 marks) What is the difference between a block-free algorithm and a wait-free algorithm? Of the two, which kind of algorithm did the specification of Assignment 1 ask for?

A wait-free algorithm is both block-free and starvation free. So, a wait-free algorithm can guarantee each process can enter critical section within finite steps which block-free only specifies that it is possible to enter if we put the time horizon infinitely long.
Assignment 1 asked for a block-free algorithm.

(c) (2 marks) The performance analysis in Section 4 found no situation where Michael and Scott's two-lock algorithm outperforms lock-free algorithms. What's the benefit of the two-lock algorithm?

It can accommodate the machines that don't support multiprogramming and universal atomic primitive such as *compare_and_swap or load_linked/store_conditional*.
It also outperforms the single lock algorithm when several processes are running at the same time.

(d) (5 marks) The compare-and-swap (CAS) on line E17 of the non-blocking algorithm has the accompanying comment "Try to swing Tail to the inserted node". This suggests that, if the CAS fails, an enqueue operation can terminate even if its work is unfinished. Is this a problem? Explain why or why not.

No, because the node has already been inserted into the list. The condition when E17 fails is that other processes have already inserted new nodes to the tail so the variable $tail$ is no long the last node in the queue. Also, even Q->Tail is not the last node in the queue, it would be corrected in line E8 and E13.

Question 6 (25 marks)

(a) (4 marks) What is the ABA problem? Explain informally and in your own words.

The ABA problem occurs when the shared variable is changed from A to B and back to A later. The process that read A before would think that the environment hasn't been changed yet because the shared variable is the same as the one it has however the fact is that other processes have already executed several times.

(b) (3 marks) How does the authors' use of modification counters help mitigate the ABA problem?

They use $compare\text{-}and\text{-}swap$ instruction for a structure that contains a modification counter and a pointer to node_t. CAS is called when modifying values so that it fails even if the values are the same when the modification counter has been changed.

(c) (5 marks) Ominously, the safety analysis in Section 3.1 is predicated upon the assumption that the ABA problem will never occur. Describe a scenario where, in the lock-free algorithm, the queue can end up corrupted if the ABA problem occurs.

Assume that the maximum value for an unsigned integer is $maxi$, for a dequeue process, the recorded $head$ is $(node_i, c_i)$. Before it executes the CAS operation in $D13$, there are already more than $maxi$ enqueues and dequeues successfully performed and the ptr recorded in Q->Head is the same as the $node_i$ recorded in variable $head$ that the process holds. This is possible because $node_i$ has been freed yet and the system allocates this piece of memory to a new node again. And this new created node with memory space $node_i$ is stored in Q->Head.ptr. In addition, the counter of Q->Head becomes $(c_i + n * maxi)\ mod\ maxi = c_i$ for all n that is a natural number. So, the processor would regard the value stored in &Q->Head and $head$ equal. Then the CAS will succeed.

However, the next.ptr that the process holds has already been freed by other process long time ago. The resulting new Q->Head is not the actual Q->Head->next.ptr rather the old next.ptr more than $maxi$ steps ago. As a result, the queue will not be properly linked anymore.

(d) (3 marks) Sometimes, ABA-type situations are innocuous. Consider the program in Table 3. Describe how the CAS at p3 may succeed, despite n having changed since the read on line p2.

p_3 may still succeed when q executes multiple of three times while p is waiting to enter p_3.

(e) (10 marks) The end result of running the program in Table 3 is unaffected by whether ABA situations happen or not. Or to be precise, the program will satisfy the following Hoare triple: {n = 0} P||Q {n = 2} Where P and Q are the left and right processes, respectively. Formulate a single global invariant that suffices to establish this Hoare triple.

Let's define two auxiliary variables k_1 and k_2. They are both initialised to 0 and k_1 is only incremented by one when CAS in p_3 succeeds and k_2 is incremented by one when CAS in q_3 is executed successfully.

Also define k_3 and k_4 that are set to x and y on line 2 respectively.

Then the invariant I is

$$\begin{aligned}
&\big(p@p_4 \Rightarrow (k_1 + i = 11) \wedge (k_1 \equiv k_2 (mod\ 3) \vee k_3 = n - 1)\big) \\
&\wedge \big(q@q_4 \Rightarrow (k_2 + j = 11) \wedge (k_1 \equiv k_2 (mod\ 3) \vee k_4 = n - 1)\big) \\
&\wedge n = (k_1 + k_2)\ mod\ 3 \\
&\wedge \big(\neg p@p_4 \Rightarrow (k_1 + i = 10)\big) \\
&\wedge \big(\neg q@q_4 \Rightarrow (k_2 + j = 10)\big)
\end{aligned}$$

The invariant ensures that CAS is succeeds only when n is not changed by another process or when another process executes multiple of three times more than itself. Thus, the final n should be the sum of number of successes, which is $20\ mod\ 3 = 2$.

Question 7 (10 marks)

The last item in Section 3.1 states "Tail always points to a node in the linked list, because it never lags behind Head, so it can never point to a deleted node".

(a) (5 marks) This statement is false for the two-lock concurrent queue. Describe how a state can be reached where Tail points to a deleted node.

When a new node linked *(Q->Tail->next executed)* but the tail Q->Tail hasn't been updated. Other processes try to empty the queue by repeatedly calling dequeue. Until only one node is remained in the queue, that is *Q->Head = Q->Tail* and new_head = Q->Tail->next. Q->Head will point to the new node that the previous enqueuer created. This time, Q->Tail lags behind Q->Head. Then, the one that Q->Tail points to will be freed by the dequeuer.

(b) (5 marks) Is it possible to reach a state where Head points to a deleted node? Explain why or why not.

No, because only dequeue can modify Q->Head and free a node. So, it guarantees that Q->Head will point to next valid node before changing Head and freeing a node.