2020-02-22 | ../../

Distributed Computing MidSem Solutions

This take-home exam permits the use of a Linux/Windows laptop/PC running javac, g++, Eclipse, Idea, and other software development and drawing tools. It is otherwise a traditional closed book, closed notes exam. In particular, you are honor bound ***not*** to Internet surf or access any content already existing (other than as indicated) once you access the exam until you turnin the answers. The primary reasons in making this a take-home are to relieve time pressure and give you a comfortable (computing/ home) environment. Do not give or take help from others.

Submit your `answers.pdf` as an email attachment to me.

My answers: (i) Are illustrative/ pedagogical – may not be the best or the shortest. (ii) Include brief commentary on how the class performed.

1. (5 points each) The following statements may or may not be (fully or partially) valid. Explain the underlined technical terms occurring in each statement. Explain/ discuss/ dispute the statement. It is *possible* to write no more than, say, five, lines each, and yet receive full score.
    1. "Some philosophers may remain hungry forever." is a safety, not liveness, property. **Answer:** Our Andrews book itself uses "bad" and "good", which cannot be defined. Many reproduced this as the answer; I marked it as not full-scoring. The crucial word-groups are {never, for-ever, always, …} and {eventually, will-happen, …}. E.g., {"A very good property P never happens."} P is a safety property. No body got a good answer.
    2. In the context of RPC, a procedure designed to be used remotely *can* use global variables to communicate with the caller.

    3. Consider the program segment given here. Determine *P* a predicate that characterizes the weakest deadlock-free precondition for the program. Also, explain how you arrived at P.

    ```
    co <await x >= 7 -> x := x - 3>
       <await x >= 6 -> x := x + 1>
       <await x >= 5 -> x := x + 1>
    oc
    ```

    Answer: You can arrive at P using whatever you have – does not have to be wp. It is useful to digest a wrong answer: x == 7. All three await-conditions are satisfied, and one will be chosen by the system. Suppose x >= 7 is chosen, and x becomes 4. Neither of the two remaining await-conditions are satisfied. So, x = = 7 is not the answer. Would x ?= 8 work? Would x ?= 6 or 5 work? You work it out. Then you have to apply the "weakest" expectation that we required.

    4. Compute *wp(S, i = 100)*, where S is

    ```
    if i < 150 then i := i + 150 else i := 100 fi
    ```

    Answer: You can guess the answer quickly, but this Q is checking if you understood the wp-mechanism. You work it out.

2. (20 points each)

1. In the distributed programming context, `P(m); Critical-Section; V(m)` is not starvation-free. Give a detailed starvation scenario. What does Morris' algorithm do to eliminate this scenario?
   Answer: Straightforward. Re-read my lecture notes.
2. Construct the ssend(pid, msg) and srecv(pid, msg) primitives of synchronous message passing from asend(pid, msg) and arecv(pid, msg) primitives of asynchronous message passing.

3. Reconstruct the binary "tree" serialized below. List each node as a triplet of (info integer, left-link and right-link), as usual. The first digit in the sequence is the offset of the root node.

```
marshalled sequence: 2 3 4 7 4 7 8 4 A 1 4 2 7
offsets in hex:      1 2 3 4 5 6 7 8 9 A B C D
```

   Answer: node@2 = (3, left offset 3, right offset D), node@4 =(7, 4, 6), node@7 = (8, 8, 9), node@A = (1, B, C). Figure TBD.

4. The *Recursive Data Representation: Small Set of Integers* of Hoare's CSP paper is reproduced below. Extend it to respond to a command `S(1)!remove(x)` from S(0) that removes x from the set, if it contains it, and does nothing if it does not.

```
S(i: 1 .. 100) ::
*[   n: integer; S(i-1)?has(n) --> S(0)!false
 []  n: integer; S(i-1)?insert(n) -->
         *[  m: integer; S(i-1)?has(m) -->
              [   m <=  n --> S(0)!(m = n)
              []  m >   n --> S(i+1)!has(m)
              ]
          [] m: integer; S(i-1)?insert(m) -->
              [  m < n --> S(i+1)!insert(n); n := m
              [] m = n -->  skip
              [] m > n --> S(i+1)!insert(m)
 ] ] ]
```

   Answer: (Just analysis and hints. Based on my lecture after this exam. Solution deliberately held back.) The CSP code above has two loops. Recall the invariant assertions of the loops. Outer Loop Assertion: Control stays in the outer loop when the S(i) process is empty handed. Once the control enters the inner loop, S(i) now holds n. It never comes out of it (because we do not yet have a remove operation).

   Placement: So, where do we place the `remove(x)`? Within the inner loop. On receiving `remove(x)` if x == n, S(i) should get out of the inner loop. But, CSP does not have a "break out of loop", so we introduce a Boolean b which remains true in the inner loop until x == n, when b is set to false. Inner Loop Assertion - Part 1: We cannot have S(i) empty handed unless all S(i+1 .. 100) are also empty handed. Inner Loop Assertion - Part 2: Values held by S(j), for j < i, are less than what S(i) holds, S(j), for j > i, are either holding values greater than what S(i) holds, or are empty-handed.

   Action to be taken: Left shift. How? Do we first ask S(i+1) are you empty handed? If yes, nothing to be done. If no, we should ask for the value it is holding. S(i+1) now changed state from not-empty to empty – so it should take the same action. Pause this, and let us go back to remove(x) reception. Why did S(i) receive it? Because S(i-1) held a number n less than x.

   For you to finish: Write all of the CSP code for this problem.

   Gotchas: Unmatched query bangs are deadlocks.

3. (0 points) [For survey purposes only.] Please record your effort in minutes for each of the above ~~ten~~ eight items. Other feedback you wish to give is also welcome.
