ID:_______________________

• Start each solution by giving the basic idea for the solution. Followed by the detailed solution.

• Close material.

• 3 hours.

• Answer all 4 questions.
ID:________________________Splitters: (25%) 

1. (a) Prove that it is impossible to wait-free implement the following splitter in the standard shared memory model in which there are only read/write atomic registers. The splitter partitions processes into two non-empty sets; It has two exits, and not all processors exit on the same exit, in runs in which more than two processors access the splitter. I.e., if \( k > 2 \) (and \( k \leq n \)) processes access the splitter then it partitions them into two non-empty sets.

(b) Implement a splitter from read/write atomic registers that partitions processes into three sets, with each set containing at most 2/3 of the arriving processes. I.e., this splitter has three exits, and on each exit at most 2/3 of the incoming processes exit. (you may assume that \( n \) is known).

(Unlike the standard splitter in the literature the splitters in a. and b. do not have a STOP or WIN exit).

(c) Consider a new object called Participating Set (PS). The PS has one operation called check-in. A process that checks-in gets as a response a snapshot of processes (set of process names) that have checked-in before or together with it. The sets returned in the responses satisfy the snapshot properties. I.e., let \( S_i \) be the set returned to process \( i \), then (i) \( i \in S_i \) and (ii) for every \( i, j, i \neq j \), either \( S_i \subseteq S_j \) or \( S_j \subseteq S_i \).

You have to prove the following claim: Let \( A \) be an arbitrary wait-free PS algorithm using only atomic read/write registers for \( n \) processes (\( n \) is known to the algorithm). Assume that all the processes try to check-in concurrently. Then, it must be possible to schedule them in such a way that each process returns a set which includes all others.
2. **New; \( k \)-atomic registers (30%)**

Company Bugtel built a shared memory system in which processors communicate only by reading and writing shared \( k \)-atomic-registers. The linearize specification of a Bugtel \( k \)-atomic-register, is such that (a) the reads and writes are ordered in a linearizable order and (b) a read returns the value of one of the \( k \) last write operations. In other words, for any execution, there is some way of totally ordering the overlapping reads and writes operations (the standard linearization order). Call this order, serialized-order. Then the value returned by each read is any value written by one of the \( k \) writes that precedes this read in the serialized-order. (As usual operations take effect sometimes after their invocation and before .)

**Question:** Is it possible to build a standard atomic read write register system from the Bugtel \( k \)-atomic-registers. Prove your answer, by either proving why it is impossible, or providing a construction (which may use any construction we studied in class as a building block).
3. New; Synchronous Message Passing Consensus with crash failures (25%)

In this question we consider the synchronous consensus algorithm that was presented in class (the first consensus algorithm that was presented). In that algorithm, that solves the consensus problem in a synchronous (message-passing) system in which at most \( f \) processes fail by crashing. Recall that in the round in which a processor crashes (i.e., fail-stops) only an arbitrary subset of these messages it tries to send in that round successfully reaches its destination (and after this round it stops participating in the algorithm). The time complexity of the algorithm presented was \( f + 1 \) rounds. Recall that in a crashing failure (i.e., fail-stop as we called it in class), it is required that the final decision value be the input of some process (i.e., not necessarily the input of a non-faulty process).

(a) Consider the following variant of the crashing failure model: a synchronous system in which at most \( f \) processes fail by almost-clean crashes, that is, in a round in which a processor crashes \textbf{it either sends all its messages, or all its messages except one, or none} (and after this round it stops participating in the algorithm). Design an algorithm that solves consensus as fast as possible in this model (i.e., with as few rounds as possible). How many rounds it needs? Justify and explain your answer.

(b) Now consider the following other variant of the crashing failure model: a synchronous system in which at most \( f \) processes fail by almost-clean crashes, that is, in a round in which a processor crashes \textbf{it either sends all its messages, or exactly one, or none} (and after this round it stops participating in the algorithm). Design an algorithm that solves consensus as fast as possible in this model (i.e., with as few rounds as possible). How many rounds it needs? Justify and explain your answer.
Here is a presentation of a wait free implementation of *Fifo Queue* from *fetch-and-add* and *swap* objects. The implementation is based on Herlihy and Wing’s construction from . A queue is defined by the two operations performed on it, *Enq* and *Deq*. *Enq*(Q, v) inserts v into queue Q’s head. *Deq*(Q) returns the item at queue Q’s tail. Assuming that every item in the queue is distinct, the sequential specification of the queue is as follows:

(a) If \( x = \text{Deq}(Q) \) then *Enq*(Q, x) is before that *Deq* operation.
(b) If \( x = \text{Deq}(Q) \) is before \( y = \text{Deq}(Q) \) then *Enq*(Q, x) is before *Enq*(Q, y).

The implementation uses one *fetch-and-add* object - counter - that can also be read by applying F&A(0). In addition it uses an unbounded array of *swap* objects - *items*. The *Enq* operation listed in figure 1 is simple. A process first F&A(1) to the counter. It then assigns the item into the queue by swapping its value into the appropriate entry - the counter value - in the *items* array.

The *Deq* operation starts by reading the counter current value. This value serves as the upper bound on the number of items within the queue. The process then scans the *items* array from *items*[1] up to the upper bound, by swapping null in every entry until it returns a non null value - which is the return value of the operation. The time of the *Deq* operation is therefore unbounded though it is finite. The code for the operation is listed in figure 1.

Question: What is wrong with this implementation, give a scenario that shows the incorrect behavior of the algorithm.
shared  
  
  object initialized to 0.

  array of swap

  objects initialized to null.

local  
  
  head, i, m of type integer; item of type data-item

Op Enq(item)
  
  head := F&A(counter, 1)  /* F&A returns the value after the addition

  Swap(items[head], item)

end Enq

Op Deq() Returns(item)

  m := F&A(counter, 0)

  for i := 1 to m
    
    item := Swap(items[i], null);

    if item ≠ null
      return (item);

  
end for

  return (empty);

end Deq

Figure 1: The Enq and Deq operations