EPFL

Prof. Rachid Guerraoui
Concurrent Computing 2024/2025 - CS-453
February 1, 2025, 9:15 to 12:15
CM1105, CM13

X

X

SCIPER : **X** Signature :

- This exam is "closed book": no notes, electronics, or cheat sheets are allowed.
- Unless otherwise stated, keep in mind that only one operation on one shared object (e.g., a read or a write of a register) can be executed by a process in a single step. To avoid confusion (and common mistakes), write only a single atomic step in each line of an algorithm.
- Remember to write which variable represents which shared object (e.g., registers).
- Unless otherwise stated, we assume atomic multi-valued MRMW shared registers.
- Unless otherwise stated, we ask for *linearizable* and *wait-free* algorithms.
- Unless otherwise stated, we assume an *asynchronous* system of n processes $\{p_1, ..., p_n\}$ which might crash.
- Arrays which appear in algorithms in the text are indexed starting from 0.

- Make sure that your name and SCIPER number appear on **every** loose sheet of paper you hand in.
- You are **only** allowed to use additional pages handed to you (available upon request).

Good luck!

Problem	Max Points	Score
1	5	
2	4	
3	3	
4	6	
5	3	
6	4	
Total	25	

Question 1 *This question is worth 5 points.*

Consider the following pseudocode for a SRSW register R, which uses a base SRSW register `REG`. This question studies how the properties of `REG` (i.e., whether it is safe or regular) affect the properties of R (i.e., whether it is atomic).

```
1  Shared state:
2  A (base) multi-valued SRSW register REG
3
4  Local state:
5  Var sn ← 0, last_sn ← 0, last_val ← ⊥
6
7  operation R.write(v)
8      sn ← sn + 1
9      REG.write((sn, v))
10     return
11
12 operation R.read()
13     (REG_sn, REG_v) ← REG.read()
14     If (REG_sn > last_sn) then
15         last_sn ← REG_sn
16         last_val ← REG_v
17     return last_val
```

Figure 1: Pseudocode of atomic register R.

1. **(1 point)** Explain the difference between (the specification of) a safe register and a regular register.
2. **(2 points)** Suppose that the base register `REG` used above is *safe*. Does R implement an atomic register? (If yes, justify why. Otherwise, provide an example that violates atomicity.)
3. **(2 points)** Suppose that the base register `REG` used above is *regular*. Does R implement an atomic register? (If yes, justify why. Otherwise, provide an example that violates atomicity.)

Please answer inside the box !

Question 2 *This question is worth 4 points.*

Consider an example of a transactional memory system. For simplicity, assume that this system supports only the following two transactions:

```
1  Shared state:
2  An integer array Balances[M], initially [1000, 0, ..., 0]. // All accounts have a balance of 0 except the first
3
4  transaction 1 (i, j, v)
5      if (0 ≤ i < M and 0 ≤ j < M and 0 ≤ v and Balances[i] ≥ v) then
6          Balances[i] ← Balances[i] − v
7          Balances[j] ← Balances[j] + v
8      return
9
10 transaction 2 (i)
11     if (0 ≤ i < M) then
12         acc ← 0
13         for (k from 0 to M − 1) do
14             acc ← acc + Balances[k]
15         Total ← acc
16         Division ← Balances[i]/Total
17         return Division
18     else
19         return ⊥
```

Figure 2: A transactional memory system with two types of transactions.

All shared memory variables are read in the order they appear in the transaction.

1. **(1 points)** Briefly explain *serializability* (a criterion of transactional memory).
2. **(1 points)** Briefly explain *opacity* (a criterion of transactional memory).

For each of the following two cases, indicate what value(s) can the variable *Total* in transaction 2 take, and if there are any errors that can occur, justifying your answer:

3. **(1 point)** Case 1: the above transactional memory system ensures *opacity*.
4. **(1 point)** Case 2: the above transactional memory system ensures serializability but not *opacity*.

Please answer inside the box !

Question 3 *This question is worth 3 points.*

Consider the atomic *commit-adopt* object, which has the following specification. Every process p proposes an input value v to such an object and obtains an output, which consists of a pair (dec, val); dec can be either *commit* or *adopt*. The following properties are satisfied:

- **Validity:** If a process obtains output (commit, v) or (adopt, v), then v was proposed by some process.
- **Commitment:** If every process that proposes does so with the same value, then no process may output (adopt, v) for any value v.
- **Agreement:** If a process p outputs (commit, v) and a process q outputs (commit, v') or (adopt, v'), then $v = v'$.
- **Termination (Wait-freedom):** Every correct process eventually obtains an output.

Consider the following incorrect implementation of an atomic *commit-adopt* object from atomic wait-free snapshot objects:

```
1  Shared state:
2  Snapshot objects A ← [⊥, ⊥, ··· , ⊥], B ← [⊥, ⊥, ··· , ⊥]
3
   /* The identifier of the process is i (starting from 0)                    */
4  procedure propose_i(v)
5      est ← v
6      A.update(i, est)
7
8      a ← A.snapshot()
9      if all non-⊥ values in a are est then
10         B.update(i, [true, est])
11     else
12         B.update(i, [false, est])
13
14     b ← B.snapshot()
15     if all non-⊥ values in b are [true, ∗] then
16         return (commit, est)
17     else
18         return (adopt, est)
```

Figure 3: An incorrect implementation of the commit-adopt object.

1. **(1 point)** What properties of the commit-adopt object are violated? For each violated property, explain why it is violated (providing an example of an incorrect execution).
2. **(2 points)** Indicate a fix for the above implementation such that it correctly implements the commit-adopt object. Prove that every property is now satisfied.

Please answer inside the box !

Question 4 *This question is worth 6 points.*

1. **(1 point)** Explain the difference between obstruction-freedom, lock-freedom, and wait-freedom.
2. **(1 point)** Give the definition for the agreement and validity properties of consensus.

Please answer inside the box !

Consider the following implementation of a consensus object for 2 processes using snapshot objects and commit-adopt objects (see Question 3 for the specification of the commit-adopt object).

```
1  Shared state:
2  An infinite array of snapshot objects, S[∞], each initialized to [⊥, ⊥]
3  An infinite array of commit-adopt objects, C[∞].
4
5  procedure propose_i(v)
6      est ← v
7      r ← 0
8      loop:
9          S[r].update(i, est)
10         s ← S[r].snapshot()
11
           /* Picks the maximum non-⊥ value from s                              */
12         est ← max(s)
13         (dec, est) ← C[r].propose(est)
14         if dec = commit then
15             return est
16
17         r ← r + 1
```

Figure 4: An implementation of an obstruction-free consensus object.

3. **(1 point)** Prove that the above implementation correctly implements obstruction-free consensus.
4. **(1 point)** Prove that the above implementation does not implement wait-free consensus. (If necessary, you can directly state results taught in class.)

Please answer inside the box !

For this part only, assume that both processes have access to an independent *coin* object that provides a boolean flip() operation. The *coin*.flip() operation randomly returns 1 with probability 1/2 and 0 with probability 1/2.

5. **(2 points)** Modify the loop of the obstruction-free consensus algorithm above such that the probability of returning at the end of each loop iteration is at least 1/2. Write and justify your answer below.

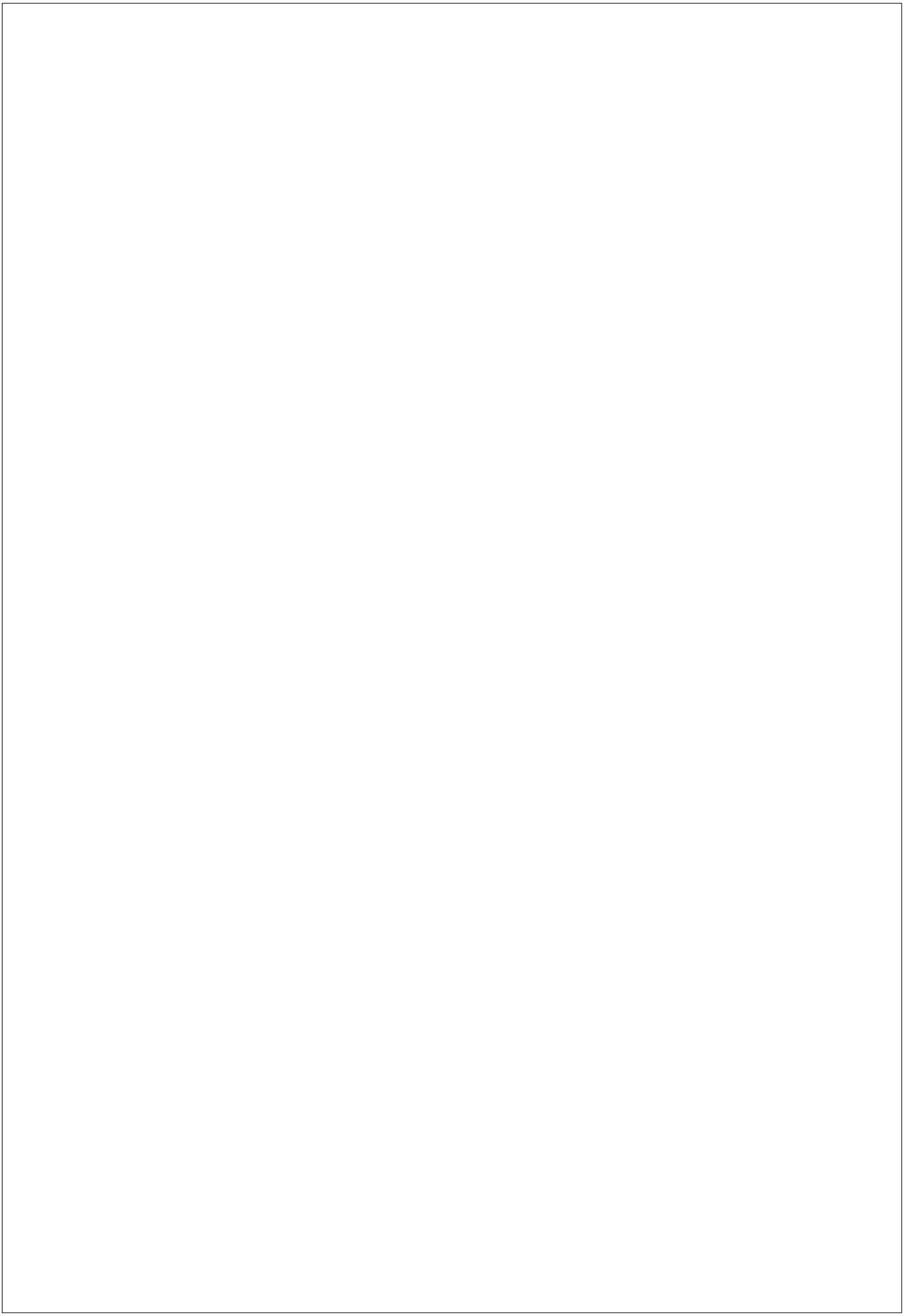
Please answer inside the box !

Question 5 *This question is worth 3 points.*

Recall the *Leader*$\Diamond$ object seen in class, which provides a leader() operation that returns a boolean.

1. **(1 point)** What are the properties of the *Leader*$\Diamond$ object?
2. **(2 points)** Implement wait-free consensus using the *Leader*$\Diamond$ object and any number of atomic snapshot objects and atomic MRMW registers. (If necessary, you may reference/reuse code from previous questions.)

Please answer inside the box !



Question 6 *This question is worth 4 points.*

1. **(2 points)** Consider a linearizable and wait-free SWAP object that supports the swap and read operations. The sequential specification of the SWAP object is shown below:

```
1 Shared state:
2 Register R initialized to 1 (of size 2 bits).
3
4 procedure swap()
  /* This next line swaps the bits of register R (using logical one-bit shifts)        */
5     R ← (R << 1) + (R >> 1)
6     return
7
8 procedure read()
9     return R
```

Figure 5: Sequential specification of a SWAP object.

What is the consensus number of the SWAP object? Provide a proof.

(If necessary, you can directly state results taught in class.)

2. **(2 points)** Assume a SWAP object that additionally supports a wait-free decrement operation (shown below). Show that this new object has a consensus number of at least 2. (You can set the initial value of register R.)

```
1 Shared state:
2 Register R (of size 2 bits).
3
4 procedure swap()
5     R ← (R << 1) + (R >> 1)
6     return
7
8 procedure read()
9     return R
10
11 procedure decrement()
12    R ← R − 1
13    return
```

Figure 6: Sequential specification of the SWAP object that furthermore supports a decrement operation.

For register R in figures 5 and 6 you can assume we use two's complement representation (i.e., $-2_{10} = 10_2$, $-1_{10} = 11_2$, $0_{10} = 00_2$, and $1_{10} = 01_2$). The decrement (dec) operation thus applies the following transition: $1_{10} \overset{dec}{\to} 0_{10} \overset{dec}{\to} -1_{10} \overset{dec}{\to} -2_{10} \overset{dec}{\to} 1_{10} \overset{dec}{\to} \cdots$ (equivalent in binary: $01_2 \overset{dec}{\to} 00_2 \overset{dec}{\to} 11_2 \overset{dec}{\to} 10_2 \overset{dec}{\to} 01_2 \overset{dec}{\to} \cdots$)

Please answer inside the box !