The F-snapshot Problem.

Gal Amram
Department of Computer Science,
Ben-Gurion University, Beer-Sheva, Israel
galamra@cs.bgu.ac.il

Abstract

Aguilera, Gafni and Lamport introduced the signaling problem in [3]. In this problem, two processes numbered 0 and 1 can call two procedures: update and Fscan. A parameter of the problem is a two-variable function $F(x_0, x_1)$. Each process p_i can assign values to variable x_i by calling update(v) with some data value v, and compute the value: $F(x_0, x_1)$ by executing an Fscan procedure. The problem is interesting when the domain of F is infinite and the range of F is finite. In this case, some "access restrictions" are imposed that limit the size of the registers that the Fscan procedure can access.

Aguilera et al. provided a non-blocking solution and asked whether a wait-free solution exists. A positive answer can be found in [5]. The natural generalization of the two-process signaling problem to an arbitrary number of processes turns out to yield an interesting generalization of the fundamental snapshot problem, which we call the F-snapshot problem. In this problem n processes can write values to an n-segment array (each process to its own segment), and can read and obtain the value of an n-variable function F on the array of segments. In case that the range of F is finite, it is required that only bounded registers are accessed when the processes apply the function F to the array, although the data values written to the segments may be taken from an infinite set. We provide here an affirmative answer to the question of Aguilera et al. for an arbitrary number of processes. Our solution employs only single-writer atomic registers, and its time complexity is $O(n \log n)$.

1 Introduction

In this paper we introduce a solution to the F-snapshot problem, which is a generalization of the well-studied snapshot problem (introduced independently by Afek et al. [1], by Anderson [6] and by Aspnes and Herlihy [7]). The snapshot object involves n asynchronous processes that share an array of n segments. Each process p_i can write values to the i-th segment by invoking an update procedure with a value taken from some range of values: $Vals$, and can scan the entire array by invoking an instantaneous scan procedure. For any function $F : Vals^n \to D$ (where D is any set and $Vals^n$ is the set of n-tuples of members of $Vals$) the F-snapshot variant differs from the snapshot problem in that the Fscan operation has to return the value $F(v_0, \dots, v_{n-1})$ of the instantaneous segment values $v_0, \dots, v_{n-1}$. That is in comparison to the standard scan operation, which returns the vector of values that the segments store at an instantaneous moment.

The F-snapshot problem is interesting only if we impose an additional requirement, without which it can be trivially implemented by applying the function F (assumed to be computable) to the values returned by the standard scan operation. This additional requirement, for the case $n = 2$, was suggested by Aguilera, Gafni and Lamport [3] (see also [2]) in what they called there the signaling problem. Thus, our F-snapshot problem is a generalization of both the standard snapshot problem and the signaling problem (generalizing this problem from the $n = 2$ case to the general case of arbitrary n).

In the signaling problem, the set $Vals$ is possibly infinite and the range of F, D is finite (and small). It is required that an Fscan operation uses only bounded registers. That is, registers that can store only finitely many different values (the update operations may access unbounded registers). The signaling problem was formulated just for two processes in [3], and a wait-free solution for this problem was left there as an open problem. Thus, solving the general F-snapshot sets quite a challenge. A wait-free solution to the signaling problem is given in [5], and here we present a wait free solution to the general F-snapshot problem.

In [3], the signaling problem is justified for efficiency reasons. We consider a case in which the processes write values to their segments taken from an infinite range, but they are interested in some restricted data regarding these values (for example, which process invoked the largest value, how many different values there are etc.). An F-snapshot implementation may be more efficient in these cases than a snapshot implementation, since it is not necessary to scan the entire array for extracting the required information, and it suffices to read only bounded registers. Efficiency is mostly guaranteed when the Fscan operations are likely to be invoked much more frequently than the update procedures.

Moreover, the authors of [3] showed that a signaling algorithm can be used to solve the mailbox problem which is the main problem that [3] deals with. With a similar approach, a signaling algorithm can be also used to implement a solution to the N-buffer problem [23] (see also [5] for further discussion). At the mailbox problem, a processor and a device communicate through an interrupt controller and it is required that the processor will know if there are some unhandled requests, only by reading bounded registers. At the N-buffer problem, a producer sends messages to a consumer, through a message buffer of size N, and they need to check if the buffer is empty, full or neither-empty-nor-full by reading bounded registers.

In the same way, a solution to the F-snapshot problem can be used to implement a generalized mailbox algorithm, in which there are several devices, and the processor can check which devices are waiting for its response by reading only bounded registers. Similarly, an F-snapshot algorithm can be used to implement a generalized N-buffer in which there are possibly many producers and many consumers[1].

Now we describe the F-snapshot problem formally. Let $P = \{p_0, \dots, p_{n-1}\}$ be a set of n-asynchronous processes that communicate through shared registers and let $F : Vals^n \longrightarrow D$ be an n-variable computable function from a (possibly infinite) domain $Vals$, into $D = Rng(F)$. The problem is to implement two procedures:

1. update(v) - invoked with an element $v \in Vals$. This procedure writes v to the i-th segment of an n-array A, when invoked by p_i.

2. Fscan - returns a value $d \in D$. This procedure returns $F(A[0], \dots, A[n-1])$, in contrast to a scan procedure which returns the values stored at the entire array: $(A[0], \dots, A[n-1])$.

The implementation needs to satisfy the following requirements:

[1] Assuming that the queue of messages supports enqueue and dequeue operations by several processes.

$\mathsf{update}(v)$

1. $A[i] := v$

$\mathsf{Fscan}()$

1. return $F(A[0], \ldots, A[n-1])$

Figure 1: The F-snapshot atomic implementation, code for process p_i

1. All procedures are wait free. That is, each procedure eventually returns, if the executing process keep taking steps.
2. If D is finite, then only bounded registers are accessed during Fscan operations.
3. Only single-writer multi-reader atomic registers are employed.

The reader may note that as the domain of F could be infinite, the update procedure must access also unbounded registers.

For correctness of F-snapshot implementations, we adapt the well known Linearizability condition, formulated by Herlihy and Wing [16]. Roughly speaking, an F-snapshot algorithm is correct if for any of its executions the following hold: Each procedure execution can be identified with a unique moment during its actual execution (named the linearization point), such that the resulting sequential execution belongs to a set of correct sequential executions: the sequential specification of the object. The sequential specification of the F-snapshot object includes all executions of the atomic implementation, presented in Figure 1. The code uses an array $A[0..n-1]$.

In this paper, we present a solution to the F-snapshot problem. Each operation in our algorithm consists of $O(n \log n)$ actions addressed to the shared registers. The rest of the paper is organized as follows: Preliminaries are given in Section 2. In Section 3 we present our algorithm and explain the ideas behind it. Section 4 explains briefly how to construct a linearization of an execution, while a detailed correctness proof is given at the Appendix. Conclusions are given in Section 5.

Related Work

As been explained, the F-snapshot problem generalizes the signaling problem, from the case that there only two processes to an arbitrary number of processes. In [3], Aguilera et al. presented the signaling problem and gave a non-blocking solution. A wait free algorithm is given in [5].

Jayanti presented the f-array object which assumes n processes that write values to m multi-writer registers, $v_1, \ldots, v_m$ and compute an m-variable function f on the values that the registers store [21]. However, while in this paper we seek for a linearizable implementation for the F-snapshot object from single-writer registers in which the Fscan operation accesses bounded registers, the scope of [21] is different. Jayanti presents in [21] an implementation for the f-array object from registers and an LL/SC object. The motivation of [21] is to improve complexity measures, when assumptions on the function f implies that it is not necessary to scan the entire array for the computation of f.

2 Preliminaries

2.1 The Model of Computation

The model of computation in this paper is standard. We assume n asynchronous processes $p_1, \ldots, p_n$ that communicate through shared single-writer registers. Thus, each registers is "owned" by a unique process. At an individual step, a process may write to one of its single-writer registers and perform an internal computation, or read a register and perform an internal computation that depends on the value it read.

An F-snapshot implementation provides each process with a code for the update and Fscan procedures. At an execution, each process executes update and Fscan operations in some arbitrary order, and the update

operations are invoked with arbitrary data values from the set *Vals*. Formally, an execution τ is a finite or infinite sequence of atomic actions (also named low-level events) that the processes perform while executing update and Fscan operations. As the processes are asynchronous, for each process p_i, there is no bound on the number of steps taken by other processes in-between two consecutive actions performed by p_i. In particular, a process may crash and take no additional steps at an execution.

At an execution τ, each atomic action is a part of a unique Fscan or update procedure execution, executed by some process. A set of actions that correspond to a procedure execution is named an operation, or an high-level event. An high-level event may be complete if the executing process executed all the procedure instructions and returned, or pending otherwise.

The low-level events at an execution are linearly ordered by the precedence relation $<$. The precedence relation $<$ is naturally extended over high-level events: We write $A < B$ if A is complete and for every $a \in A$ and $b \in B$, $a < b$. Similarly, we relate high-level events with low-level events. For a low-level event e and high-level event A, we write $e < A$ if $e < a$ for every $a \in A$ and we write $A < e$ if A is complete and $a < e$ for every $a \in A$. Incomparable events are said to be concurrent.

2.2 Linearizability

An execution is linearizable if the high-level events can be identified with instantaneous moments during their executions so that this identification yields a correct sequential execution. More precisely, it is required to find such linearization points for all complete high-level events, where pending operations may be omitted or artificially completed.

In some cases the linearization points can be identified with an execution of some fixed instruction. When no such fixed linearization points exist, it is convenient to define the linearizability criterion in an equivalent manner. We say that an execution τ is linearizable, if there is a linear ordering $(\mathcal{H}, \prec)$ such that

1. $\mathcal{H}$ includes all complete high-level events and some pending high-level events.
2. $\prec$ extends $<$ over $\mathcal{H}$.
3. $(\mathcal{H}, \prec)$ belongs to the sequential specification. Namely, $\prec$ is a precedence relation over $\mathcal{H}$ obtained by some execution of the algorithm in Figure 1.

An implementation is linearizable if all its executions are linearizable.

As an Fscan operation needs to return a value obtained by applying F on the values that the segments store, it is required to assume some initial value for each segment. For convenience, we assume that at the beginning of each execution, each process executes an initial update event invoked with an initial data value. These events write initial values to the registers and variables and they precede all other high-level events.

3 The F-snapshot Algorithm

First we explain the main ideas behind the algorithm. The reader may want to consider our explanations, while examining the code of the algorithm given in Figure 3, and its local procedures in Figure 4.

The crucial obstacle for solving the problem is that an Fscan procedure cannot access unbounded registers, but it is required to apply the function F on values that are stored in unbounded registers. To overcome this issue, we apply the function F on the values that the segments store, during an execution of an update operation. When process p_i performs an update operation invoked with a data value *val*, it writes *val* into a snapshot object V (line 2), scans this snapshot object (line 3), applies the function F on the view it obtained and stores the outcome in a local variable *ans* (line 4). Then, before it returns, the process writes the outcome it obtained into a snapshot object named *Flags* (line 16). A process executing an Fscan operation, scans the snapshot object *Flags* and it needs to choose the most up-to-date value among the values suggested by the processes. We need to provide the *Flags* object with an additional information, so that the executing process could decide correctly which value to return. However, this additional information needs to be taken from a finite range due to the problem limitations.

As a first attempt, one may suggest to use bounded concurrent timestamps (see [10],[11],[12]). Bounded timestamps are used traditionally to label writes, but here we actually need to label scans. Namely, we need

to know in what order the scans of the snapshot object V occurred. If we will try to label the updates of $Flags$, then a process executing an Fscan operation can mistakenly rely on the order between the labeling operations and not on the order between the scan events addressed to the snapshot object V. Similarly, labeling the writes to V will not work either by the same reason. We see that the approach of using bounded timestamps, at least in its simplest form, will not succeed.

3.1 The Classify Mechanism

For determining the ordering between scan events addressed to V, we adapt the common technique of counting update events [9],[8],[19]. When a process performs an update operation, it increases a counter (line 1) and writes this counter to V together with the data value with which the update operation was invoked (line 2). When process p_i scans V (line 3), it sums these counters to obtain a natural number that reflects how recent its view is (line 9). This approach resembles the snapshot algorithm presented by Israeli, Shaham and Shirazi [19]. They used this technique to implement a snapshot algorithm in which the time complexity of the scan procedure is $O(n)$. In their construction, while executing a scan operation, the executing process returns the view of the process that presents the latest activity, reflected by the largest sum.

Since an Fscan operation cannot access unbounded registers, we cannot adopt the discussed approach as it was used in [19]. In our algorithm, the reading of these natural numbers is done within the update operation. The process writes the sum it obtained into a snapshot object named $ViewSum$ (lines 10,12), and scans $ViewSum$ to compare its view with the views obtained by the other processes. Afterward, it classifies all other processes into two categories: $winners$ - the processes that posses a later view, reflected by a largest sum, and $losers$ - processes with outdated view. This is done by calling the local classify procedure, when processes id's are used for breaking symmetry. These sets of $winners$ and $losers$ are stored at the segment $Flags[i]$ (lines 15,16).

3.2 The coloring Mechanism

When a process p_k executes an Fscan operation, it scans the $Flags$ array and it tries to extract the most up-to-date view, referring to the fields $Flags[i].winner$ and $Flags[i].losers$ for $i = 0, \ldots, n-1$. For any pair of processes p_i and p_j, p_k tries to understand which process's view is more recent. The problem is that the processes may provide contradicting information. As an example, the process may find that $j \in Flags[i].winners$ (which means that p_i thinks that p_j's view is more up-to-date than its view), but it is possible that also $i \in Flags[j].winners$. Namely, it is possible that both p_i and p_j think that the other process knows better.

The coloring mechanism ensures that the problem described above can occur only in some "typical" executions (with which our next mechanism deals). The update events by each process alternate between 3 possible colors: 0, 1 or 2 (line 1). Each process posses a three-field variable, in correspondence to the three colors, named *myview*. After a process sums the counters it sees (lines 3,9), it writes the sum it obtained into *myview*[*color*] (line 10) and deletes data obtained in its second-previous update operation (line 11) to erase confusing information. Then, it writes the value that *myview* stores into $ViewSum$ (line 12). Now, when process p_i scans $ViewSum$, in each segment $ViewSum[j]$, it finds two integers. These are the sums that p_j computed in its two previous update events. When p_i executes its local classify procedure, it also writes the color it saw. For example, it writes (j, c) to $winners$ for $c \in \{0, 1, 2\}$, if it reads from $ViewSum[j][c]$ a number larger than the sum it obtained in line 9 (when id's are taken into account for breaking symmetry). Each process p_i writes the values of its local sets $winners$ and $losers$ to $Flags[i]$, together with the color of the update operation it is executing.

Coming back to our example, now each process also specifies the color of the update operation it saw. If process p_k executes an Fscan operation and it finds that $(j, c) \in Flags[i].winners$, then it understands that p_i saw in $ViewSum[j][c]$ an integer larger than the number it obtained. However, if it sees that $Flags[j].color \neq c$, it just disregards p_i's information.

3.3 Adding Bounded Timestamps

The coloring mechanism does not prevent entirely the possibility that processes will provide contradicting information. Assume for example, that while executing an Fscan operation, p_k finds that $Flags[i].color = c_i$, $Flags[j].color = c_j$, $(j, c_j) \in Flags[i].winners$ and $(i, c_i) \in Flags[j].winners$. Thus, both p_i and p_j claim that the other process is more up-to-date. When such a situation occurs, one of the processes provides reliable information. This is the process that scanned $ViewSum$ later before updating $Flags$.

When such a situation occurs, the processes use timestamps to inform which process is trustworthy. We use a simple timestamps system in which the timestamps are vertices of a nine-vertices directed graph $G = (V_G, E_G)$. A detailed explanation can be found in chapter 2 of [15], or in [18]. The graph G consists of three cycles, each cycle includes three vertices. In addition, there is an edge from each vertex at the i-th cycle to each vertex at the $i-1 \pmod 3$ cycle. Formally, $V_G = \{(i,j) : i,j \in \{0,1,2\}\}$, and there is an edge from $v = (i_1, j_1)$ to $u = (i_2, j_2)$ if $i_1 = i_2$ and $j_1 = j_2 + 1 \pmod 3$, or $i_1 = i_2 + 1 \pmod 3$, see Figure 2 for illustration. The vertices of G are named timestamps, and if $(v, u) \in E$ we say that v dominates u and we write $u <_{ts} v$. Intuitively, v dominates u means that the timestamp v represents a later moment than the timestamp u.

We note that there are no cycles of length two in G. In addition, for any two timestamps v, u, we can find a timestamp w that dominates both v and u. We take a function $next : V_G \times V_G \longrightarrow V_G$ that satisfies this property. That is, for any timestamps v, u: $v <_{ts} next(v, u)$ and $u <_{ts} next(v, u)$.

Any process p_i holds n pairs of timestamps. Each pair consists of a new timestamp and an old timestamp. These pairs are stored in a snapshot object named VTS. When p_i executes an update operation it scans VTS (line 5). Then, against each process p_j it chooses a timestamp that dominates the pair of timestamps it read from $VTS[j][i]$, using the function $next$. p_i stores the timestamp it obtained as its new timestamp, keeps its former timestamp available as its old timestamp and updates VTS (consider lines 5-8 and the local procedure newts). Finally, p_i stores its n-vector of pairs of timestamps in $Flags[i]$ while updating the $Flags$ object (line 16).

Now, we consider again the situation in which process p_k executes an Fscan operation, and finds that two processes p_i and p_j, provide contradicting information as described earlier. In this case, the scanner checks the timestamps that the processes present. The process that its new timestamp dominates the other process's new timestamp is the reliable one. More precisely, the scanner considers the timestamps $Flags[i].vts[j].new$ and $Flags[j].vts[i].new$. The information provided by the process with the later timestamp is the right information. These timestamps are used only when processes provide contradicting information. In other cases the timestamps do not necessarily reflect the right ordering between the processes' views.

3.4 The Code

Now we specify the code of the algorithm and we start by presenting the data structures and the type of the registers and variables. First, the algorithm uses four snapshot objects.

1. V - each entry $V[i]$ stores a pair: $(n, val) \in \mathbb{N} \times Vals$. val is the value with which the update procedure is invoked, and $n \in \mathbb{N}$ counts the number of update operations invoked by p_i. Note that as these values are taken from an infinite range, an Fscan operation cannot access this object. Initially each segment stores the value $(0, x_0)$ for some fixed $x_0 \in Vals$.

2. VTS - each entry $VTS[i]$ stores an n-array: $vts_i[0..n-1]$ of pairs of timestamps. Thus, each entry $vts_i[j] = (v, u)$ where v, u are timestamps. The first field is denoted $vts_i[j].old$, while the second field is $vts_i[j].new$ i.e. $(v, u) = (vts_i[j].old, vts_i[j].new)$. Initially, each field $VTS[i][j]$ stores (v_0, v_0) for some fixed $v_0 \in V_G$.

3. $ViewSum$ - each entry $ViewSum[i]$ is a triple: $viewsum_i[0..2]$ of natural numbers when each entry may also store $null$. That is, $ViewSum[i] \in (\mathbb{N} \cup \{null\}) \times (\mathbb{N} \cup \{null\}) \times (\mathbb{N} \cup \{null\})$. There are three fields in correspondence to three possible colors of the update operations. The initial value of each segment $ViewSum[i]$ is $(0, null, null)$.

4. $Flags$ - this is the bounded object scanned during Fscan operations. Each entry $Flags[i]$ stores an element of type $flag$. The $flag$ type consists of five fields:

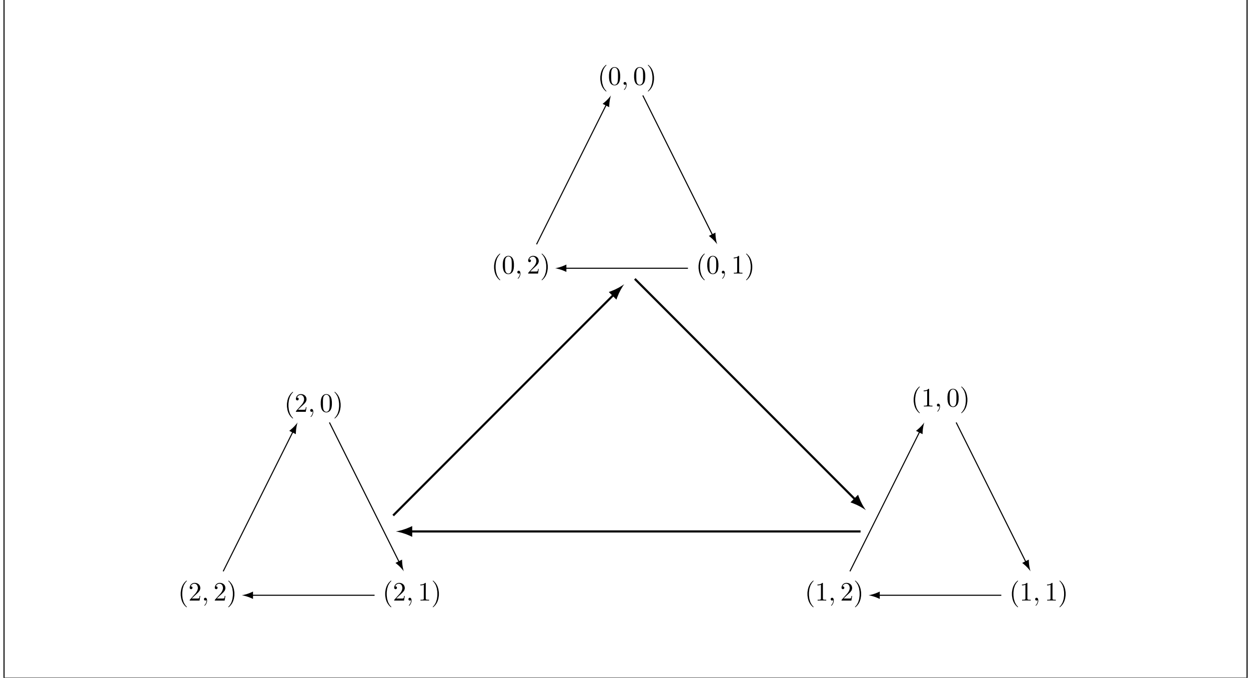

Figure 2: The graph G. Thick arrows denote edges from each node at the first triangle to each node at the second triangle

(a) $flag.color \in \{0, 1, 2\}$. Initially this field is 0.

(b) $flag.vts$ - an n-array of pairs of timestamps. Initially all pairs are (v_0, v_0). Recall that (v_0, v_0) is also the initial value of each $VTS[i][j]$.

(c) $flag.winners$, $flag.losers$ - sets of pairs of the form: $(i, c) \in \{0, \dots, n-1\} \times \{0, 1, 2\}$. At the i-th segment $flag.winners$ and $flag.losers$ are initialized to $\{(j, 0) : i < j\}$ and $\{(j, 0) : j < i\}$ respectively.

(d) $flag.ans \in D$. The initial value of this field is $F(x_0, x_0, \dots, x_0)$. Recall that $x_0 \in Vals$ is the initial value of each entry $V[i]$.

Each process uses several local variables:

1. $color \in \{0, 1, 2\}$. Initially $color = 0$.

2. $counter, viewsum \in \mathbb{N}$. The initial value of these variables is 0.

3. $val \in Vals$.

4. $ans \in D$.

5. $myview \in (N \cup \{null\}) \times (N \cup \{null\}) \times (N \cup \{null\})$. Initially $myview = (0, null, null)$.

6. $winners, losers$ - sets that include elements from the range $\{0, \dots, n-1\} \times \{0, 1, 2\}$.

7. $vts_i[0..n-1]$ - an n-array of pairs of timestamps.

8. $ts.old, ts.new$ - timestamps.

9. Other variables that are used for storing information while scanning the snapshot objects (lines 3,5,13). The type of each such variable is in correspondence to the type of the objects that are scanned.

The algorithm uses four local procedures:

```
update(val)

1. counter := counter + 1, color := counter mod 3,
2. V.update(counter, val)
3. (v_0, ..., v_{n-1}) = V.scan
4. ans := F(v_0.val, ..., v_{n-1}.val)
5. (vts_0, ..., vts_{n-1}) := VTS.scan
6. for j = 0 to n - 1 do
7.     vts_i[j] := newts(vts_j[i], vts_i[j])
8. VTS.update(vts_i)
9. viewsum := v_0.counter + ··· + v_{n-1}.counter
10. myview[color] := viewsum
11. myview[color + 1 (mod 3)] := null
12. ViewSum.update(myview)
13. (view_0, ..., view_{n-1}) := ViewSum.scan
14. classify(view_0, ..., view_{n-1})
15. flag := newflag()
16. Flags.update(flag)
```

```
Fscan()

1. (flag_0, ..., flag_{n-1}) := Flags.scan
2. winner := find_max(flag_0, ..., flag_{n-1})
3. return flag_winner.ans
```

Figure 3: code for p_i

1. classify - gets n triples of natural numbers as arguments. Each triple represents the amount of knowledge that the corresponding process has obtained in its recent update operations. As each update operation has a color from the set $\{0, 1, 2\}$, each entry is a triple in correspondence to three possible colors. This procedure constructs the sets $flag.winners$ and $flag.losers$ based on the considerations explained above.

2. newflag. Creates a new flag before updating the $Flags$ object.

3. newts - gets two pairs of timestamps: $pair_1, pair_2$ and returns a pair of timestamps, $pair_3$ such that $pair_3.new$ dominates both fields of $pair_1$, and $pair_3.old = pair_2.new$

4. find_max - gets n $flags$ as arguments and returns an element from $\{0, \ldots, n-1\}$. This procedure is invoked during an Fscan operation and the element that this procedure returns is the id of the most up-to-date process.

The procedures classify, newflag and newts are presented in Figure 4. The procedure find_max is discussed in the next subsection.

```
classify(view_0, ..., view_{n-1})

1. winners :=
     {(j, c) : view_j[c] > viewsum} ∪
     {(j, c) : view_j[c] = viewsum ∧ i < j}
2. losers :=
     {(j, c) : view_j[c] < viewsum} ∪
     {(j, c) : view_j[c] = viewsum ∧ i > j}
```

```
newflag()

1. flag.color := color
2. flag.vts := vts_i
3. flag.winners := winners
4. flag.losers := losers
5. flag.ans := ans
6. return flag
```

```
newts((u, v), (u', v'))

1. ts.old := v'
2. ts.new = (next(u, v))
3. return (ts.old, ts.new)
```

Figure 4: Local procedures

3.5 The Procedure find_max

This procedure is invoked during an execution of an Fscan event S, and it returns the id of the most up-to-date process. Thus, the process that executes S returns the value $flag_i.ans$ in case that find_max returns i.

The find_max procedure of an Fscan operation S gets n flags as arguments: $flags(S) := (flag_0, \ldots, flag_{n-1})$. The procedure returns a maximal element in relation $<_S \subseteq \{0, \ldots, n-1\} \times \{0, \ldots, n-1\}$ that we define here. The relation $<_S$ is defined by reference to $flags(S)$ in Definition 2.

Definition 1. *Let p_i and p_j be two processes and write: $flag_i.color = c_i$ and $flag_j.color = c_j$. We say that p_i and p_j are in conflict, if one of the following occurs:*

1. *$(j, c_j) \in flag_i.winners$ and $(i, c_i) \in flag_j.winners$.*
2. *$(j, c_j) \in flag_i.losers$ and $(i, c_i) \in flag_j.losers$.*

Definition 1 is important since, as we shall prove, for each two processes p_i, p_j and an Fscan event S, the $flag$ of one of these processes determines correctly the ordering between p_i and p_j. That is, if p_i is the reliable process and if (for example) $(j, c_j) \in flag_i.winners$ and the color in p_j's $flag$ is c_j, than p_j is indeed more up-to-date than p_i (more precisely, the ans field of p_j's flag is more up-to-date) as indicated by p_i's flag. The problem is that we do not know which process provides correct information among any pair of processes. However, this problem does not arise when the processes are not in conflict. When processes provide contradicting information we use the processes' timestamps to find the trustworthy process.

Definition 2. *Let p_i and p_j be two processes and write: $flag_i.color = c_i$, $flag_j.color = c_j$. $i <_S j$ if one of the following occurs:*

1. *p_i and p_j are not in conflict and $(i, c_i) \in flag_j.losers$.*
2. *p_i and p_j are not in conflict and $(j, c_j) \in flag_i.winners$*
3. *p_i and p_j are in conflict, $flag_i.vts[j].new <_{ts} flag_j.vts[i].new$ and $(i, c_i) \in flag_j.losers$.*
4. *p_i and p_j are in conflict, $flag_j.vts[i].new <_{ts} flag_i.vts[j].new$ and $(j, c_j) \in flag_i.winners$.*

An element $i \in \{0, \ldots, n-1\}$ is maximal in $<_S$ if there is no $j \neq i$ such that $i <_S j$. The procedure find_max$(flag_0, \ldots, flag_{n-1})$ (line 2) returns a maximal element in $<_S$ (we shall prove that such a maximal element exists in any Fscan event). This procedure accesses only local variables and we omit the technical but easy implementation of this procedure.

4 Correctness

Fixing an execution τ of our algorithm, we need to show that the precedence relation defined over the high-level events in τ, $<$ can be extended into a linear ordering, $\prec$ that belongs to the sequential specification of the F-snapshot object. Here we explain briefly how to construct a linearization of τ (namely, how to define $\prec$). A detailed proof is given at the Appendix.

First, we linearize all update events. The linearization point of an update operation is the execution of line 2, namely the update of the snapshot object A. For defining the linearization points this way, we assume that the execution of the command in line 2 is atomic. The sufficient of this assumption is discussed at the Appendix.

Now, we linearize each Fscan event S, immediately after some update event. Let S be an Fscan operation. Assume that the find_max procedure in S returns j (note that it should be proved why find_max returns some element, since we have not argued yet that relation $<_S$ admits a maximal element). Let U_j be the p_j-update operation that wrote to $Flags$ the value read in S. For each $i < n$, let U_i be the p_i-update event that wrote to V the value read in U_j. Note that $val(S) = F(val(U_0) \ldots, val(U_{n-1}))$. Let U_k be the update event that includes the latest write to V among $\{U_0, \ldots, U_{n-1}\}$. We linearize S immediately after U_k.

Finally, if we linearized several scan events $S_1, \ldots, S_m$ immediately after the same update event U, we order these events in some arbitrary way that extends $<$ over $\{S_1, \ldots, S_m\}$.

5 Conclusions

We present the F-snapshot problem which generalizes the signaling problem introduced in [3], from the two-process case to an arbitrary number of processes. We described a wait-free F-snapshot algorithm that employs only single-writer registers, and proved its correctness. Our algorithm uses four snapshot objects. For efficiency, we can use the snapshot implementation by Attiya and Rachman [9] for these objects. Thus, the time complexity of our algorithm is $O(n \log n)$. Since the $Flags$ object is accessed during Fscan operations, it is required to use the bounded version of the algorithm in [9] (described in Section 4.4). As the F-snapshot problem generalizes the snapshot problem, where F is chosen to be the identity function, the F-snapshot problem inherits the linear time lower bound of the snapshot problem. The $O(n)$ lower bound holds for the Fscan procedure [22] and also for the update procedure [20].

It is known that the snapshot object can be implemented with time complexity $O(n)$ when multi-writers are allowed as Inoue, Masuzawa, Chen and Tokura proved [17]. Inoue et al. present an algorithm that solves the lattice agreement problem. Then, the reduction by Attiya, Herlihy and Rachman [8], provides a linear snapshot implementation with multi-writer registers. However, this reduction requires unbounded registers (and even unbounded memory). Hence, the F-snapshot limitations forbid using this implementation for the $Flags$ snapshot object in our algorithm. Therefore, the question if there is a linear F-snapshot implementation using multi-writer registers is not answered here, although there is a linear snapshot implementation that uses multi-writer registers.

The main idea behind our algorithm is, in some sense, orthogonal to the classical bounded timestamps problem. In a time-stamp system, the processes label their writes to an array of data values. These labels provide a linear-ordering that extends the actual partial-ordering between the writes to the array. In our algorithm, we use bounded data to label the ordering between scan events and not between write events. We are not aware of a formulation of this abstract problem. This is a possible direction for further research, influenced by ideas behind our algorithm.

By the essence of the F-snapshot problem, an interesting complexity measure is the size of the bounded registers that are accessed during an Fscan operation (named "flags"). For convenience, we assume that the range of F, $D = \{0, \ldots, |D|-1\}$ and we note that $\Omega(\log |D|)$ is a trivial lower bound for the size of each flag. In our algorithm, each register writes to the snapshot object $Flags$ a data value of type $flag$. This data type consists of several fields when the largest are the sets: $winners$ and $losers$ that require $O(n)$ bits, and the field ans stores elements from D thus can be assumed to be consist of $\log |D|$ bits. However, the $Flags$ implementation require additional fields when the largest one stores a view: n-tuple of values of type $flag$. Therefore, the size of each flag in our algorithm is $O(n^2 + n \log |D|)$. We believe that this can be significantly improved.

In our algorithm, the Fscan procedure accesses only bounded registers due to the problem constrains and the update procedure accesses unbounded registers (otherwise the problem is unsolvable). The segments of the snapshot object V store elements from $Vals$ (which might be infinite), and counters that infinitely grow. Hence, if the function F has a finite domain, the update procedure will still access unbounded registers. Thus, in those cases, it is better to use some other implementation such as the bounded version of the algorithm in [9]. An interesting question that arises is whether there is a F-snapshot algorithm that satisfies both properties:

1. If F has a finite range, then the Fscan procedure accesses only bounded registers.
2. If F has a finite domain, then only bounded registers are accessed.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890, September 1993.

[2] M. K. Aguilera, E. Gafni, L. Lamport. The mailbox problem (Extended Abstract). In *Distributed Computing*, pages 1-15, 2008.

[3] M. K. Aguilera, E. Gafni, and L. Lamport. The mailbox problem. *Distributed Computing*, 23(2): 113-134, 2010.

[4] R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *Logic in Computer Science (LICS)*, pages 219-228, 1996.

[5] G. Amram. On the signaling problem. In *International Conference on Distributed Computing and Networking*, pages 44-65, 2014.

[6] J. Anderson. Composite registers. *Distributed Computing*, 6(3):141-154, 1993.

[7] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Architectures and Algorithms*, pages 340-349, 1990.

[8] H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3): 121-132, 1995.

[9] H. Attiya and O. Rachman. Atomic snapshots in $O(n \log n)$ operations. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 29-40, 1993.

[10] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible. In *Proc. of 21st STOC*, pages 454-466, 1989.

[11] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM J. Comput.*, 26(2):418-455, 1997.

[12] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible!. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 655-666, 1992.

[13] F. Ellen, Y. Lev, V. Luchangco and M. Moir. SNZI: scalable nonzero indicators. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 13-22. ACM, 2007.

[14] R. Guerraoui and E. Ruppert, Linearizability Is Not Always a Safety Property. In *Proceeding of the 2nd international conference, Networked Systems*, pages 57-69, 2014.

[15] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, NY, USA, 2008.

[16] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463-492, 1990.

[17] M. Inoue and W. Chen. Linear-time snapshot using multi-writer multi-reader registers. In *WDAG 94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 130-140, 1994.

[18] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4):205-209, 1993.

[19] A. Israeli, A. Shaham, and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory*, 28(5):469-486, 1995.

[20] A. Israeli and A. Shirazi. The time complexity of updating snapshot memories. *Information Processing Letters*, 65(1):33-40, 1998.

[21] P. Jayanti. f-arrays: Implementation and applications. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 270-279, 2002.

[22] P. Jayanti, K. Tan and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438-456, 2000.

[23] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125-145, 1977.

Appendix

Fixing an execution τ of our algorithm, we need to show that the precedence relation defined over the high-level events in τ, $<$ can be extended into a linear ordering, $\prec$ that belongs to the sequential specification of the F-snapshot object. In Section 3.5, a relation $<_S$ was defined. An Fscan event S returns a value stored in $Flags[j].ans$ where j is maximal in relation $<_S$. Thus, for proving correctness we also need to show that for any Fscan event in τ, S, $<_S$ admits a maximal element.

τ may be finite or infinite. However, it suffices to prove linearizabilty for the case that τ is finite. This holds since linearizability of deterministic objects is a safety property as proved by Guerraoui and Ruppert [14]. Since the algorithm is wait-free, we may also assume that all operations in τ are complete. Indeed, if there are pending operations in τ, we can let the processes take additional steps and complete their pending operations. This way, an execution that extends τ is obtained, and a linearization of the resulting execution admits a linearization of τ as well.

The execution τ is a sequence of atomic actions addressed to the shared memory. Thus, the procedure executions addressed to the snapshot objects (e.g line 2 of the update procedure) are not atomic and represent a sequence of instructions. However, by using a linearizable implementation for the snapshot objects, we may assume for convenience that all the operations addressed to the snapshot objects are atomic. This assumption simplifies our proof since we do not need to speak about the linearization points of these operations and the corresponding extension of $<$. For further discussion about using linearizable implementations see [4] and [16].

As explained in Section 2.2, we assume n initial update events by the processes, where the initial update event by process p_j is denoted I_j. These initial high-level events write initial values to the registers and they precede all other events.

Each atomic action is either a read action (also named read event) or a write action (also named write event). If e is a read event from a register R, then $val(e)$ is the value that the executing process read from R in e. Similarly, if e is a write event, $val(e)$ is the value written in e. For an update(v) operation, U we write $val(U) = v$, and for an Fscan operation S, $val(S)$ is the value returned in S. The value of each initial update event is x_0. That is, for every process id j, $val(I_j) = x_0$. Recall that the initial value of the entry $V[j]$ is $(0, x_0)$. Each atomic action belongs to a unique operation. For a low-level event e, $[e]$ is the high-level event that includes e. Therefore, $e \in [e]$.

Our algorithm employs several snapshot objects. Thus, for preventing confusion, we use the notation A.update and A.scan to denote invocations of update and scan procedures addressed to object A. Note that an A.update(x) invocation by p_i writes x to the i-th segment of A.

The following notations are important in our proof:

1. For an A.scan event e addressed to a snapshot object A, we define $\mu_j(e)$ to be the maximal A.update event by p_j that precedes e. Thus, $val(e)[j] = val(\mu_j(e))$ for any A.scan event e.

2. Let U be an update event and let i be a process id. If U is an initial update event we set $\alpha_i(U) = I_i$, the initial p_i-update event. Otherwise, $\alpha_i(U)$ is the p_i-update event in which p_i wrote to $V[i]$ the value that was read from $V[i]$ in U. That is:

$$\alpha_i(U) = [\mu_i(V.\mathsf{scan}(U))]$$

   where V.scan(U) is the (unique) V.scan event in U, which corresponds to the execution of line 3 in the code of the update procedure.

3. Let S be an Fscan event in which $winner = j$ (the invocation of find_max in S returns j). Let e be the (unique) $Flags$.scan event in S, and let $U_j = [\mu_j(e)]$. For a process id i, we define $\alpha_i(S) = \alpha_i(U_j)$.

4. Let S be an Fscan event. For a process id i, $\beta_i(S)$ is the p_i-update event that wrote to $Flags[i]$ the value read in S. That is, $\beta_i(S) = [\mu_i(Flags.\mathsf{scan}(S))]$ where $Flags$.scan(S) is the (unique) $Flags$.scan event in S.

Two easy observations that will be useful later are the following:

Lemma 3. *For each p_i-update event U, $\alpha_i(U) = U$.*

Lemma 4. *Let U_1 and U_2 be two* update *events such that $U_1 < U_2$. Then, for each process id i, $\alpha_i(U_1) \leq \alpha_i(U_2)$.*

Lemma 3 holds since the V.update operation in U precedes the V.scan operation (lines 2 and 3). Lemma 4 holds since the V.scan event in U_1 precedes the V.scan event in U_2.

We fix an Fscan event S and we shall prove that there is a maximal element in relation $<_S$. For each process id i, write $U_i = \beta_i(S)$, $flag_i = val(\mu_i(Flags.\mathsf{scan}(S)))$, and $c_i = flag_i.color$. That is, U_i is the p_i-update event that wrote to $Flags$ the value read in S, $flag_i$ is the value that p_i wrote to $Flags[i]$ in U_i and c_i is the value of the color field of $flag_i$. According to the initial values of the snapshot object $Flags$ and by the code of the classify procedure, the following hold:

Lemma 5. *For a pair $(j, c) \in \{0, \ldots, n-1\} \times \{0, 1, 2\}$ and a process p_i, at most one of the following occurs:*

1. *$(j, c) \in flag_i.winners$.*
2. *$(j, c) \in flag_i.losers$.*

Corollary 6. *If $i <_S j$, then $\neg(j \ <_S i)$.*

Proof. Consider Definition 2, and observe that relation $<_S$ over i and j is determined only by the $flag$ of one of the processes p_i and p_j. Hence, this is a consequence from the previous lemma. ☐

If U_i is not the initial update event I_i, when p_i executed U_i, it computed a natural number while executing line 9. Let m_i denote this number. If $U_i = I_i$, define $m_i = 0$. We argue that m_i reflects correctly how recent p_i's view is.

Lemma 7. *For two processes p_i and p_j, if $(m_i, i) < (m_j, j)$ at the lexicographic order, then for each process id k, $\alpha_k(U_i) \leq \alpha_k(U_j)$.*

Proof. If $U_i = I_i$, then for each process id k, $\alpha_k(U_i)$ is the first p_k-update event and the lemma holds. If $U_j = I_j$, then $m_j = 0$ which implies that $m_i = 0$. Thus, $U_i = I_i$ and we are done. It is left to deal with the case that $U_i \neq I_i$ and $U_j \neq I_j$.

Towards a contradiction, assume that $\alpha_k(U_j) < \alpha_k(U_i)$ for some process id k. We conclude that the V.scan event in U_i occurred after the V.scan event in U_j. Therefore, the counter that p_i read from each field $V[t].counter$ is larger than the counter that p_j read (note that the l-th update operation by each process writes l to this field). Hence, $m_j \leq m_i$. However, since the integer that p_i read from $V[k].counter$ is strictly larger than the one that p_j read, $m_j < m_i$ in contradiction to the assumption that $(m_i, i) < (m_j, j)$. ☐

During the execution of S, for each two processes p_i and p_j, the process that executes S decides whether $i <_S j$ or $j <_S i$. The decision is made upon the values of these processes' $flags$. The next lemmas show that for each such a pair of processes, at least one of these processes' $flags$ provides reliable information. That is to say, for some process (say, p_i) the following occurs:

- If $flag_i.ans$ is more up-to-date than $flag_j.ans$, then $(j, c_j) \in flag_i.losers$.
- If $flag_i.ans$ is less up-to-date than $flag_j.ans$, then $(j, c_j) \in flag_i.winners$.

Lemma 8. *Let p_i and p_j be two processes such that $U_i \neq I_i$. Let $e_i \in U_i$ be the* update *of $ViewSum$ in U_i (line 12) and let e_j be the* update *of $ViewSum$ in U_j. Let e be the* scan *event of $ViewSum$ in U_i (line 13). If $e_j < e_i$, then one of the following holds:*

1. *$\mu_j(e) = e_j$ or,*
2. *$\mu_j(e) = e' > e_j$ and there is no p_j-*update *event between $U_j = [e_j]$ and $[e']$.*

Proof. Since $e_j < e_i$ and since (by the code) $e_i < e$, we see that $e_j < e$ and hence, $e_j \leq \mu_j(e)$. Thus, we need to show that there is at most one $ViewSum$.update event by p_j between e_j and e.

Assume for a contradiction that e' and e'' are two $ViewSum$.update events by p_j such that

$$e_j < e' < e'' < e.$$

Each $ViewSum$.update event belongs to a unique update event so there are two different p_j-update operations $U' = [e']$ and $U'' = [e'']$. Recall that $[e_j] = U_j$ and observe that:

$$\beta_j(S) = U_j < U' < e'' < e.$$

Now, the $Flags$.scan event in S occurs after e, so it reads the value written to $Flags[j]$ in U' or in a later event. We have:

$$\beta_j(S) = U_j < U' \leq [\mu_j(Flags.\mathsf{update}(S))]$$

in contradiction to the definition of $\beta_j(S)$. □

We conclude:

Lemma 9. *Let p_i and p_j be two processes such that $U_i \neq I_i$. Let $e_i \in U_i$ be the* update *of $ViewSum$ in U_i, let e_j be the* update *of $ViewSum$ in U_j, and let e be the* scan *event of $ViewSum$ in U_i. If $e_j < e_i$, then p_i reads m_j from $ViewSum[j][c_j]$ in e, and in addition:*

1. *If $(m_i, i) < (m_j, j)$ (at the lexicographic order), then $(j, c_j) \in flag_i.winners$.*
2. *If $(m_i, i) > (m_j, j)$ (at the lexicographic order), then $(j, c_j) \in flag_i.losers$.*

Proof. $e \in U_i$ is the $ViewSum$.scan event in U_i by p_i. By the previous lemma, since $e_j < e_i$ there is at most one write to $ViewSum$ between e_j and e. Thus, the value that p_j wrote to $ViewSum[j][c_j]$ (which is m_j) has not been "deleted" (consider lines 10-12 of the update procedure). p_i reads m_j from $ViewSum[j][c_j]$ and the lemma follows from the code of the classify procedure. □

So far, we have proved that for any two processes p_i and p_j, one of these processes' $flags$ provides reliable information. Namely, the process that wrote later to $ViewSum$ during the update events U_i and U_j. The next lemma easily stems.

Lemma 10. *Let p_i and p_j be two processes which are not in conflict (the definition is given in Section 3.5). If $(m_i, i) < (m_j, j)$ lexicographically, then $i <_S j$.*

Proof. First assume that $m_j = 0$. In this case, $U_j = I_j$ is the initial update event. In addition, since $(m_i, i) < (m_j, j)$, also $m_i = 0$ thus $U_i = I_i$ as well. We conclude that $i < j$ and according to the initial values of the registers we get that $(j, 0) \in Flags[i].winners$ and $(i, 0) \in Flags[j].losers$. In addition, $c_i = c_j = 0$ and hence, $i <_S j$ as required.

Now assume that $m_j > 0$, and hence $U_j \neq I_j$. Write $e_i \in U_i$ - the update of $ViewSum$ in U_i and respectively, e_j is the update of $ViewSum$ in U_j. Assume w.l.o.g. that $e_j < e_i$ and observe that U_i is not the initial event either. By Lemma 9, $(j, c_j) \in flag_i.winners$. Since the processes are not in conflict the claim holds. □

Our next goal is to prove the same for the case that the processes are in conflict. If the processes are in conflict, we know by the previous lemmas that one of them provides reliable information. Recall that in this case, the definition of $<_S$ is according to the $flag$ of the process that presents a later timestamp. We need to show that the process with the later timestamp is the reliable one, namely the one that wrote later to $ViewSum$.

Lemma 11. *Let p_i and p_j be two processes. Let $e_i \in U_i$ be the* update *of $ViewSum$ in U_i, let e_j be the* update *of $ViewSum$ in U_j, and assume that $e_j < e_i$. If p_i and p_j are in conflict, then $flag_j.vts[i].new <_{ts} flag_i.vts[j].new$.*

Proof. First, note that $U_i \neq I_i$. Indeed, if $U_i = I_i$ we get that also $U_j = I_j$ (since $e_j < e_i$) which implies that the processes are not in conflict.

For the rest of the proof we assume that $U_j \neq I_j$. If $U_j = I_j$, then similar (and simpler) argument can be applied. Let s_j be the scan of $ViewSum$ in U_j. By Lemma 9, p_i reads m_j from $ViewSum[j][c_j]$ in U_i, but since p_i and p_j are in conflict, we conclude that p_j read some $k \neq m_i$ from $ViewSum[i][c_i]$ in s_j. Hence,

$$\mu_i(s_j) \neq e_i. \tag{1}$$

Since $e_j < e_i$ and (by the code) $e_j < s_j$, either $e_j < s_j < e_i$ or $e_j < e_i < s_j$. We claim that the former occurs and $s_j < e_i$. Assume otherwise and use Equation 1 to conclude that $e_i < \mu_i(s_j)$. Note that there can be at most one $ViewSum$.update event by p_i that follows e_i and precedes s_j (consider the arguments in the proof of Lemma 8), and hence s_j reads from $ViewSum[i]$ the value of this event. However, by the code of the update procedure, the update operation by p_i that follows U_i also writes m_i to $ViewSum[i][c_i]$. Thus, if $e_i < s_j$, then p_j reads m_i from $ViewSum[i][c_i]$ in s_j, and this is in contradiction to the assumption that the processes are in conflict. We conclude that $e_j < s_j < e_i$.

Now we claim that there is a p_i-$ViewSum$.update event between s_j and e_i. Indeed, assume not and let e' be the last $ViewSum$.update event by p_i that precedes e_i. By our assumption we have $\mu_i(s_j) = e'$. e' belongs to the last p_i-update event that precedes U_i and hence the color of this update event is $c_i - 1 \pmod 3$. Therefore, $val(e')[c_i] = null$. We conclude that p_j read $null$ from $ViewSum[i][c_i]$ in s_j and this contradicts the fact that p_i and p_j are in conflict.

We see that there is a $ViewSum$.update event by p_i between s_j and e_i. Let e'_i denotes this event. Hence,

$$s_j < e'_i < e_i.$$

Write $[e'_i] = U'$, a p_i-update event and note that $U' < U_i$. Therefore, $e'_i < U_i$ and hence

$$s_j < U_i.$$

Now, let $t_j \in U_j$ be the (unique) VTS.update event in U_j (line 8 in the code) and write $val(t_j)[i] = (x, y)$. Observe that since $t_j \in U_j$, (x, y) is also the value of $flag_j.vts[i]$. Let $s_i \in U_i$ be the (unique) VTS.scan event in U_i (line 5) and note that since $e'_i < U_i$, $e'_i < s_i$. By the code and by our conclusions we have: $t_j < s_j < e'_i < s_i$ thus

$$\mu_j(s_i) \geq t_j.$$

Note that there is at most one VTS.update event by p_j between t_j and s_i since otherwise, we would have $\beta_j(S) \neq U_j$. Furthermore, if there is such an event, it writes to $VTS[j][i]$: (y, z) for some vertex $z \in V_G$ (consider the newts code). Let (a, b) denotes the value of $VTS[i][j]$ before the execution of U_i.

Case 1. $\mu_j(s_i) = t_j$ and hence p_i reads in s_i from $VTS[j][i]$: (x, y). Thus, p_i writes in U_i to $Flags[i].vts[j]$: newts$((x, y), (a, b)) = (b, next(x, y))$. Since $(next(x, y), y) \in E_G$, $flag_j.vts[i].new <_{ts} flag_i.vts[j].new$ as required.

Case 2. $\mu_j(s_i) > t_j$ and hence p_i reads in s_i from $VTS[j][i]$: (y, z). In this case p_i writes in U_i to $Flags[i].vts[j]$: newts$((y, z), (a, b)) = (b, next(y, z))$. Since also $(next(y, z), y) \in E_G$, $flag_j.vts[i].new <_{ts} flag_i.vts[j].new$. We see that the lemma holds in this case as well.

□

The previous lemma shows that if two processes are in conflict and their $flags$ provide contradicting information, the $flag.vts$ fields determined correctly which among the two processes is the reliable one. The conclusion is that relation $<_S$ determines correctly which process presents the most up-to-date view in its $flag.ans$ field.

Lemma 12. *Let p_i and p_j be two processes. Then, $(m_i, i) < (m_j, j) \iff i <_S j$.*

Proof. First assume that $(m_i, i) < (m_j, j)$ and we shall prove that $i <_S j$. If p_i and p_j are not in conflict, then this is the case of Lemma 10. If p_i and p_j are in conflict, let e_i be the update of $ViewSum$ in U_i and let e_j be the update of $ViewSum$ in U_j. Assume w.l.o.g. that $e_j < e_i$. By Lemma 9, $(j, c_j) \in flag_i.winners$. By the previous lemma $flag_j.vts[i].new <_{ts} flag_i.vts[j].new$ thus by definition, $i <_S j$.

Now, for the other direction, assume that $i <_S j$. If $(m_j, j) < (m_i, i)$, then we get that also $j <_S i$ in contradiction to Corollary 6. Thus, $(m_i, i) < (m_j, j)$ as required. □

It is easy to see that any two update events U and U', are comparable in $\leq_\alpha$. For verifying this observation, assume w.l.o.g. that the V.scan event in U' occurs after the V.scan event in U. Clearly, $U \leq_\alpha U'$ in this case. Therefore, we conclude:.

Corollary 13. *There is a unique maximal element in $<_S$ and hence, the* find_max *procedure in S returns some $j < n$.*

Now we are ready to show that τ is a linearizable. We define a linear ordering $\prec$ on the set of all high-level events in τ. First, we define $\prec$ over the update events. Then, we define $\prec$ between update and Fscan events and finally, we define $\prec$ over Fscan events.

1. For two update operations U, U', we set $U \prec U'$ if the write to V in U precedes the write to V in U'. That is, the executions of the V.update operations are the linearization points of the update events.

2. Let S be an Fscan event. For each update event U, we decide if $U \prec S$ or $S \prec U$ by choosing an update event to linearize S immediately after it.

   For each process p_i, write $U_i = \alpha_i(S)$. We linearize S immediately after the update events $U_0, \ldots, U_{n-1}$. More precisely, we linearize S after the maximal element in $\prec$ over the set $\{U_0, \ldots, U_{n-1}\}$.

3. It is left to define $\prec$ over the Fscan events. First, we consider pairs of Fscan events S, S' such that S was linearized after an update event U and S' was linearized after an update event U' such that $U \prec U'$. In this case, we set $S \prec S'$.

   Now, for each update event U, we take the Fscan events linearized immediately after U, $S_1, \ldots, S_m$, and we define $\prec$ over these events in some arbitrary way that extends $<$ over $S_1, \ldots, S_m$.

It is easy to verify that $\prec$ is a linear ordering. We show now that $\prec$ extends $<$. Consider two high-level events $A < B$. We shall prove that $A \prec B$. The claim is trivial when A and B are update events as these events were linearized in correspondence to an execution of some fixed atomic instruction. We need to deal with the cases that A and B are both Fscan events, or one of them is an Fscan event and the other is an update event.

Case 1. $A = U$ an update event, say by p_i and $B = S$ an Fscan event. For each process p_k, let U_k denote the p_k-update event that wrote to $Flags[k]$ the value read in S. i.e. $U_k = \beta_k(S)$. Note that $U \le U_i$. Assume that the procedure find_max in S returned j thus $i \le_S j$ and $\alpha_i(S) = \alpha_i(U_j)$. Use Lemmas 12, 7 and 3 to observe that:

$$U \le U_i = \alpha_i(U_i) \le \alpha_i(U_j) = \alpha_i(S).$$

Recall that S was linearized after $\alpha_i(S)$ and hence, since $\prec$ extends $<$ over update events,

$$U \preceq U_i = \alpha_i(U_i) \preceq \alpha_i(U_j) \prec S.$$

Case 2. $A = S$ an Fscan event and $B = U$ an update event, say by p_i. Note that since $S < U$, $U \neq I_i$ the initial p_i-update event. For showing that S is linearized before U, we need to show that for each process p_k, $\alpha_k(S) \prec U$.

Assume that the procedure find_max in S returns j and write $U_j = \beta_j(S)$. For a process p_k, write $U_k = \alpha_k(S) = \alpha_k(U_j)$. If $U_j = I_j$, then $U_k = I_k$ and then it is clear that $U_k \prec U$ as required. Otherwise, let e_k be the V.update event in U_k, let s be the V.scan event in U_j and let e be the V.update event in U. Obviously, $e_k < s$. Since $\beta_j(S) = U_j$, $\neg(S < s)$, but since $S < U$, we conclude that $s < e$. As a result, $e_k < e$ which implies that $U_k \prec U$ as required.

Case 3. $A = S$ and $B = S'$ are both Fscan event. For proving that S is linearized before S' we show that for each process p_i, $\alpha_i(S) \preceq \alpha_i(S')$. Assume that the procedure find_max in S returns j and the procedure find_max in S' returns k. Write $U_j = \beta(S)$ and $U'_k = \beta_k(S')$. Hence, $\alpha_i(S) = \alpha_i(U_j)$ and $\alpha_i(S') = \alpha_i(U'_k)$. Write $U'_j = \beta_j(S')$ and use lemmas 12 and 7 to conclude that $\alpha_i(U'_j) \le \alpha_i(U'_k)$. Since $S < S'$, $U_j \le U'_j$ thus by Lemma 4 we get,

$$\alpha_i(S) = \alpha_i(U_j) \le \alpha_i(U'_j) \le \alpha_i(U_k) = \alpha_i(S).$$

Hence $\alpha_i(S) \le \alpha_i(S')$ and $\alpha_i(S) \prec \alpha_i(S')$ follows.

It is left to prove that the properties of the sequential specification are satisfied. It is easy to see that each Fscan event S returns $F(val(\alpha_0(S)), \ldots, val(\alpha_{n-1}(S)))$. Therefore, we need to verify that for each process p_i, $\alpha_i(S)$ is the maximal p_i-update event that precedes S in $\prec$. Since S was linearized after the events $\alpha_0(S), \ldots, \alpha_{n-1}(S)$, clearly $\alpha_i(S) \prec S$ for each process p_i.

Towards a contradiction, assume that for some process p_i, $U \neq \alpha_i(S)$ is the maximal p_i-update event that precedes S in $\prec$. Hence,

$$\alpha_i(S) \prec U \prec S.$$

We conclude that there is a process p_k such that

$$\alpha_i(S) \prec U \prec \alpha_k(S)$$

since otherwise, S would have been linearized before U. Note that $\alpha_k(S) \neq I_k$. Assume that the procedure find_max in S returns j and write $U_j = \beta_j(S)$. Thus, $\alpha_k(S) = \alpha_k(U_j)$. Since $\alpha_k(S)$ is not the initial p_k-event, also $U_j \neq I_j$.

Write $\alpha_i(S) = \alpha_i(U_j) = U_i$ and $\alpha_k(S) = \alpha_k(U_j) = U_k$. Let e_i be the V.update operation in U_i, let e be the V.update operation in U and let e_k be the V.update operation in U_k. Since $U_i \prec U \prec U_k$, we have

$$e_i < e < e_k.$$

Now, let s be the V.scan event in U_j. By definition, $\mu_k(s) \in U_k$ thus $\mu_k(s) = e_k$ and in particular

$$e_k < s.$$

As a result, $e_i < e < s$ thus $\mu_i(s) \neq e_i$ in contradiction to $\alpha_i(U_j) = U_i$.