Takehome: Atomic snapshots

Uri Abraham*

July 27, 2014

Abstract
We describe the single-writer registers atomic snapshot algorithm of [2] and the multiple writer registers snapshot algorithm of [4].

1 Atomic snapshots

There are two types of snapshot data structures: the single-writer and the multiwriter register snapshots. We begin with the single-writer which is simpler. There are \( N \) processes \( P_1, \ldots, P_N \). Each process \( P_i \) has a memory address \( A_i \) on which it is the only writer and which any process can read. The processes can execute two sorts of operations: Update and Scan. Process \( P_i \) executes Update\(_i\)(\( v \)) operations where \( v \) is a parameter of type Data. This operation sets atomically the value of \( A_i \) to \( v \). Any process can execute Scan operations. This operation returns an array of length \( N \) of Data values which is the list of values of all memory addresses \( A_1, \ldots, A_N \). For any Update operation \( U \) we denote with val\((U)\) the value of the parameter \( v \) with which \( U \) is invoked, and for every Scan operation \( S \), val\((S)\) is the array of length \( N \) that \( S \) returns and val\((S)[i]\) is the \( i \)-th component of that array. The linear specification is simple. The operations are linearly ordered and initial Update\(_i\)(\( v_i \)) operations are assumed. Every Scan operation \( S \) returns an array \( a = val(S) \) such that for every index \( 1 \leq i \leq N \), \( a[i] \) is the value of the last Update\(_i\) operation that precedes \( S \).

When the operations are not linearly ordered (an operation of process \( P_i \) may be concurrent with an operation of \( P_j \)) the atomicity requirement is that

---

* Models for concurrency, spring 2014
there is a linear ordering $<^*$ that extends the partially ordered precedence relation $<$ and is such that the linear specifications described above hold for $<^*$. This is the well-known atomicity of linearizability condition of [3].

In the multiwriter snapshot, a process is not associated with specific address and it may write on any address. The number of memory addresses $M$ is not necessarily equal to the number of processes $N$, and we denote with $A_1, \ldots, A_M$ the set of addresses. If $1 \leq x \leq M$ is an address index then operation $Update(x, v)$ (executed by anyone of the processes) writes the value $v$ on address $A_x$. The $Scan$ operation returns an array $(d_1, \ldots, a_M)$ of $Data$ values, where $d_x$ is the snapshot value of address $A_x$. If the operations are linearly ordered in $<$, then the linear specification is that for every $Scan$ $S$, if $val(S) = (d_1, \ldots, d_M)$ then for every address $x$ if $U$ is the last $Update$ operation on address $A_x$ that precedes $S$ then $val(U) = d_x$.

2 The Single-Writer Scan Algorithm

A variant of the algorithm of [2] is in Figure 1. Every process $P_i$ has a register $R_i$ that carries values that are triples with three entries: data, sequence-number, and report, where data is in $Data$, sequence-number is a natural number, and report is an array of length $N$ of $Data$ values. We say that such a triple is a register triple. The initial value of $R_i$ is $(v_i, 0, \emptyset)$ where $v_i$ is some assumed initial data value. We assume that for every $i$ there is an initial write action on register $R_i$ of this initial value. All later write actions on register $R_i$ are in some $Update$ execution by process $P_i$.

We clarify the difference between the memory address $A_i$ and the register $R_i$. The memory address is an abstract entity. An $Update(v)$ operation on memory address $A_i$ is implemented by reading the registers $R_j$ of the processes and writing on register $R_i$. In the single-writer algorithm, address $A_i$ is associated with process $P_i$, and so only process $P_i$ executes $Update$ operations on $A_i$ (and only process $P_i$ writes on register $R_i$). A $Scan$ operation is an implementation of an atomic read of all memory addresses $A_j$, and it is executed by reading the actual registers $R_j$ as directed by the algorithm.

Procedure $collect$ is executed by reading the registers $R_i$ for $1 \leq i \leq N$ in any order and returning the values thus collected in array $a$.

In line 0 of the $Scan$ code, procedure $collect$ is called $N + 2$ times and the sequence of arrays $a_0, \ldots, a_{N+1}$ records the results of these invocations. So the $Scan$ operation consists of $(N + 2) \times N$ read events (executed in the order
Update$(v)$ on address $A_i$:  
0. $s := \text{Scan}();$
1. $\text{seq}_i := \text{seq}_i + 1$
2. $R_i := (v, \text{seq}_i, s)$

Scan (by any process):  
0. for $k := 0$ to $N + 1$ $a_k := \text{collect}();$
1. Case 1: for some $k \in \{0, \ldots, N\}$ $a_k = a_{k+1}$
   return $(a_k[1].\text{data}, \ldots, a_k[N].\text{data});$
2. Case 2: for some $1 \leq i \leq N$ there are indexes $0 \leq m < n < N + 1$ such that $a_m[i] \neq a_{m+1}[i]$ and $a_n[i] \neq a_{n+1}[i]$
   return $a_{n+1}[i].\text{report}$

collect:  
0. for every $1 \leq j \leq N$ $a[j] := R_j;$
1. return $a;$

Figure 1: A variant of the single-writer snapshot algorithm of [2]. Operation $\text{Update}$ on memory address $A_i$ is executed by process $P_i$, and the $\text{Scan}$ operation is executed by any process.

specified). Based on the values of variables $a_0, \ldots, a_{N+1}$, the $\text{Scan}$ algorithm decides on the snapshot value to return. There are two modes by which a $\text{Scan}$ operation can return. A $\text{direct}$ return is a return via Case 1, and if Case 1 does not apply, then an $\text{indirect}$ return is via Case 2. We argue below (Lemma 2.1) that if Case 1 does not apply then it must be that Case 2 applies, so that the code tells any $\text{Scan}$ operation which value to return. In both cases a $\text{Scan}$ operation returns an array of $\text{Data}$ values, and we denote with $\text{val}(S)$ the array that $\text{Scan}$ $S$ returns.

Consider now an $\text{Update}$ operation $U$ (by $P_i$) invoked with $\text{Data}$ value $v$. We write $v = \text{val}(U)$ for the value of the parameter with which $U$ is invoked. The code for the $\text{Update}$ operation begins (in line 0) with a $\text{Scan}$ invocation whose return value is assigned to variable $s$. Thus a $\text{Scan}$ operation can be a stand-alone $\text{Scan}$ or a $\text{Scan}$ that is part of an $\text{Update}$ operation. The $\text{Scan}$ operation with which $U$ begins is denoted $\text{Scan}(S)$. After the $\text{Scan}$ execution $\text{Scan}(U)$, a local variable $\text{seq}_i$ (initialize to 0) is increased by 1. Hence the $\ell$-th $\text{Update}$ execution by $P_i$ has $\ell$ as the value of its sequence number $\text{seq}_i$. In line 2 of the $\text{Update}$ code, the value $(v, \ell, s)$ is written on register $R_i$. Here $\ell$ is the sequence number of this execution, and $s$ is the value returned by the $\text{Scan}$ operation (an array of length $N$ of $\text{Data}$ values).

**Lemma 2.1** If Case 1 does not apply to a $\text{Scan}$ operation execution $S$, then
it must be that Case 2 applies.

Proof. Suppose that a Scan operation does not return via Case 1. Then for every $k \in \{0, \ldots, N\}$ $a_k \neq a_{k+1}$. And since procedure collect returns an array of length $N$ (of register triples), $a_k \neq a_{k+1}$ implies that there is some $i = i_k$ $(1 \leq i \leq N)$ such that $a_k[i] \neq a_{k+1}[i]$. Consider the function that takes $0 \leq k \leq N$ to $i_k \in \{1, \ldots, N\}$. By the pigeon hole principle¹ there are indices $0 \leq m < n \leq N$ such that $i_m = i_n$. That is, for $i = i_m = i_n$,

$$a_m[i] \neq a_{m+1}[i] \text{ and } a_n[i] \neq a_{n+1}[i]. \quad (1)$$

So Case 2 holds.

This short argument exemplifies an idea that is central to our approach to concurrency, and although it is a very simple idea it is a cornerstone of the approach described in [1]. The idea is that the correctness proof of a distributed algorithm can be obtained in three phases.

1. The first phase establishes some properties of operation executions by a process $P_i$ that depend only on the algorithm of $P_i$ and not on the interactions of $P_i$ with the other processes or on the properties of the communication devices that the process employs.

2. The second phase combines the properties obtained in the first stage with the specifications of the communication devices employed, and obtains some abstract, higher level properties of the operation executions.

3. The third phase takes the abstract properties of the second phase and proves that they entail the desired correctness condition. Tarskian system executions serve in the third phase of the proof.

The argument given above (Lemma 2.1) to show that any Scan that does not return via Case 1 must return via Case 2, is an example of a first phase argument. The proof of Lemma 2.1 refers only to the text of the Scan algorithm, but not to the Update algorithm or to the specification of the registers used. The proof relies on the knowledge that the reads of register $R_j$ return register triple values, but not on the atomicity of these registers. In other

¹The principle says that if $f$ is a function from a finite set to a smaller set, then $f$ is not one-to-one.
words, Lemma 2.1 holds true even if the values returned by the read actions are determined randomly.

We state in the following two lemmas, without proof, all the properties that are obtained in the first phase of the proof: those that are obtained by reference to the Scan operations, and those that are obtained by reference to the Update operations. These lemmas need only local states for their proofs since no assumptions on the registers are needed.

**Lemma 2.2** Any Scan operation $S$ is either direct (i.e. returning in Case 1) or indirect (returning via Case 2).

1. If $S$ is direct then there are two collect executions $c_1 < c_2$ in $S$ such that $\text{val}(c_1) = \text{val}(c_2)$. Say $a = \text{val}(c_1)$ is this common array. The return value of $S$ is $\text{val}(S) = (a[1].\text{data}, \ldots, a[N].\text{data})$.

2. If $S$ is indirect then there is some index $1 \leq i \leq N$ and there are four read actions of register $R_i$ in $S$, $r_1 < r_2 < r_3 < r_4$, such that $\text{val}(r_1) \neq \text{val}(r_2)$ and $\text{val}(r_3) \neq \text{val}(r_4)$. Say $a = \text{val}(r_4)$. The value of $S$ in this case is $\text{val}(S) = a.\text{report}$.

**Exercise 1** In Case 2 it is possible that there are several indices for which (1) holds. How can you remove this ambiguity?

**Lemma 2.3** If $U$ is the $\ell$-th Update operation by process $P_i$, and if $\text{val}(U) = v$ is the value of $U$, then $U$ consists of a Scan execution $S = \text{Scan}(U)$ followed by a write action on register $R_i$ of the value $(v, \ell, \text{val}(S))$.

We emphasize that the proof of Lemma 2.2 depends only on the algorithm of the Scan operation, and likewise the proof of lemma 2.3 depends only on the algorithm of the Update operation. So essentially these are proofs about serial algorithms and concurrency issues do not enter in these proofs. A more elaborate discussion of this point will be given in the last section.

Now we move to the second phase of the proof. In this phase we combine the statements of lemmas 2.2 and 2.3 with the semantics of atomic registers. In fact, for simplicity we assume that all registers $R_i$ are serial, and leave to the reader to find why we are allowed to do that. So the read/write actions on the different registers are assumed to be linearly ordered by the precedence relation $<$.  

5
Exercise 2  Show that if we prove atomicity of the Scan/Update operations under the assumption that the registers are serial, then it follows that even if the registers are assumed to be atomic the desired atomicity of the snapshot algorithm follows.

If \( r \) is any read action of register \( R_i \) then \( \omega(r) \) denotes the last write action \( w \) on register \( R_i \) such that \( w < r \). Following the semantics of serial registers we get that \( \text{val}(\omega(r)) = \text{val}(r) \). It follows immediately that if \( r_1 < r_2 \) are two read actions of some serial register \( R \) then \( \omega(r_1) \leq \omega(r_2) \).

Lemma 2.4 Suppose that \( r_1 \) and \( r_2 \) are two read actions of some register \( R_i \) such that \( \text{val}(r_1) = \text{val}(r_2) \). Then \( \omega(r_1) = \omega(r_2) \).

Proof. Every write action on register \( R_i \) is either the initial write (of value \((v_i,0,\emptyset)\)) or an execution of line 2 in some Update operation by \( P_i \). If \( \omega(r_1) \neq \omega(r_2) \), then, for some \( \ell_1 \neq \ell_2 \), \( w_1 = \omega(r_1) \) is in the \( \ell_1 \)th Update operation execution and \( w_2 = \omega(r_2) \) is in the \( \ell_2 \) Update. But in this case the sequence field of \( w_1 \) is \( \ell_1 \) and the sequence field of \( w_2 \) is \( \ell_2 \), and thus \( \text{Val}(w_1) \neq \text{val}(w_2) \), which implies that \( \text{val}(r_1) \neq \text{val}(r_2) \).

Let \( S \) be a direct Scan operation (returning with Case 1). We define \( \Omega(S) \) as the sequence of Update operations whose Data values \( S \) return. In details, the definition of \( \Omega(S) \) is as follows.

Definition 2.5 Assume that \( S \) is a direct Scan operation. By Lemma 2.2 there are two collect executions \( c_1 < c_2 \) in \( S \) such that \( \text{val}(c_1) = \text{val}(c_2) \), and if \( a = \text{val}(c_1) \) then \( \text{val}(S) = (a[1].\text{data}, \ldots, a[N].\text{data}) \). For every index \( 1 \leq i \leq N \) we let \( r(i) \in c_1 \) be the read action of register \( R_i \) in \( c_1 \). Let \( w_i = \omega(r(i)) \) be the corresponding write action, and let \( W_i \) be the Update operation to which \( w_i \) belongs (or, if \( w_i \) is the initial write action on \( R_i \) then we let \( W_i \) be that initial event itself). We then define \( \Omega(S) = (W_1, \ldots, W_N) \). We shall use the notation \( \Omega_i(S) = W_i \) for \( 1 \leq i \leq N \).

In the following and in subsequent lemmas we use the notation \( \text{begin}(X) \) and \( \text{end}(X) \) to denote the moment when an operation \( X \) begins and terminates (see the lecture on moment based system-executions). When \( X \) is an Update operation execution, then \( \text{end}(X) \) can be identified with the write on register \( R_i \) in \( X \), which is its last event.

Lemma 2.6 The following properties of the function \( \Omega_i \) hold (for \( 1 \leq i \leq N \)). For any direct Scan operation \( S \):
1. \( W_i = \Omega_i(S) \) is an Update operation by \( P_i \) (or the initial write event on \( R_i \)), and \( \text{end}(W_i) < \text{end}(S) \).

\[
\text{val}(S) = (\text{val}(W_1), \ldots, \text{val}(W_N)).
\]

2. If \( V \) is any Update operation by \( P_i \) such that \( V < S \) then \( V \leq \Omega_i(S) \).

3. If \( V \) is an Update operation by \( P_j \) such that \( \text{end}(V) < \text{end}(\Omega_i(S)) \) (i is any index \( 1 \leq i \leq N \)) then \( V \leq \Omega_j(S) \).

**Proof.** Items 1 and 2 of the lemma are rather easy to prove, and so we show the proof of item 3. Let \( S \) be a direct Scan operation and suppose that \( c_1 < c_2 \) are the two collect operations in \( S \) such that \( \text{val}(c_1) = \text{val}(c_2) \) as in Lemma 2.2(1). Suppose that \( W_i = \Omega_i(S) \) and \( V \) is an Update operation by \( P_j \) such that \( \text{end}(V) < \text{end}(W_i) \) (which means that the write action in \( V \) precedes the write action in \( W_i \)). We have to prove that \( V \leq \Omega_j(S) \). By definition of \( W_i \) there is a read action \( r_i^1 \) of register \( R_i \) in \( c_1 \) such that \( \omega(r_i^1) = w_i \) is the write action in \( W_i \). By definition of \( \text{end}(W_i) \), \( \text{end}(W_i) = w_i \). Now let \( v = \text{end}(V) \) be its write action, and suppose that \( v < w_i \). Let \( r_j^2 \) be the read in \( c_2 \) of register \( R_j \). We have that \( v < w_i < r_i^1 < r_j^2 \). Hence \( v < r_j^2 \), and so \( v \leq \omega(r_j^2) \). But if \( r_j^1 \) is the read action in \( c_1 \) of register \( R_j \), then \( \text{val}(r_i^1) = \text{val}(r_j^2) \) (as \( \text{val}(c_1) = \text{val}(c_2) \)). Hence \( \omega(r_i^1) = \omega(r_j^2) \) (by Lemma 2.4) and hence \( v \leq \omega(r_i^1) \). This implies that \( V \leq \Omega_j(S) \) as desired. \( \square \)

**Exercise 3** Give a detailed example that shows that in item 3 of the lemma, \( V < \Omega_j(S) \) is possible.

Next, we analyze indirect Scan operations (i.e. those that return with Case 2). Any Scan operation \( S \) begins with a series of \( N+2 \) collect operations \( c_0, \ldots, c_{N+1} \) (an execution of line 0)\(^2\). Every collect execution contains \( N \) read actions of the registers \( R_1, \ldots, R_N \) (in any order). We denote with \( \text{begin}(S) \) the first read action in \( c_0 \), and \( \text{end}(S) \) denotes the last read action in \( c_{N+1} \). If \( S_1 \) and \( S_2 \) are Scan operations, then \( S_1 \subseteq S_2 \) means that \( \text{begin}(S_2) < \text{begin}(S_1) \) and \( \text{end}(S_1) < \text{end}(S_2) \). That is, \( S_1 \subseteq S_2 \) says that the temporal extension of \( S_1 \) is strictly included in the extension of \( S_2 \). Since any process is

\(^2\)Actually there is no need to execute all of these operations. The algorithm can stop immediately after the condition that allows Case 1 or Case 2 holds, and this is how the algorithm is originally presented.
a serial agent that executes its operations in order, if $S_1 \sqsubseteq S_2$ then $S_1$ and $S_2$ belong to different processes. And since there is a finite number of processes, relation $\sqsubseteq$ is well-founded: any descending sequence of Scan operations must be finite. Specifically, if $S_1 \sqsupseteq S_2 \sqsupseteq \cdots \sqsupseteq S_{N+1}$ is a nested sequence of length $N + 1$, then there is an index $n$ with $S_n = S_{n+1}$.

**Exercise 4** Prove this last statement.

**Lemma 2.7** Suppose that $S$ is an indirect Scan operation. Then there is an Update operation $U$ such that the following holds for $P = \text{Scan}(U)$.

1. $\text{val}(S) = \text{val}(P)$.
2. $P \sqsubseteq S$.

**Proof.** Recall that for an Update operation $U$, $\text{Scan}(U)$ is the Scan operation with which $U$ begins. By Lemma 2.2(2), there is some index $1 \leq i \leq N$ and there are four read actions of register $R_i$ in $S$, $r_1 < r_2 < r_3 < r_4$, such that $\text{val}(r_1) \neq \text{val}(r_2)$ and $\text{val}(r_3) \neq \text{val}(r_4)$. And for $a = \text{val}(r_4)$, the value of $S$ is $\text{val}(S) = a.\text{report}$. Let $w = \omega(r_4)$ be the corresponding write action on register $R_i$. Then $r_3 < w$ (or else $w < r_3$ would imply that $w = \omega(r_3) = \omega(r_4)$ in contradiction to $\text{val}(r_3) \neq \text{val}(r_4)$). There is an Update operation $U$ by $P_i$ such that $w = \text{end}(U)$. That is, $w$ is the write action of $U$. Let $P = \text{Scan}(U)$ be the Scan operation with which $U$ begins. To prove the lemma it suffices to show that $\text{begin}(S) < \text{begin}(P)$ (and $\text{end}(S) < w < r_4 \leq \text{end}(S)$ is obvious). Suppose on the contrary that $\text{begin}(P) < \text{begin}(S)$. Then we have that

$$\text{begin}(P) < r_1 < r_2 < r_3 < w.$$ 

Since the last write on register $R_i$ that precede $w$ must be before $P$, this relation implies that $\omega(r_1) = \omega(r_2)$ which is in contradiction to $\text{val}(r_1) \neq \text{val}(r_2)$.

**Exercise 5** Give detailed proofs for the following statements that were made in the proof of the lemma.

1. Why the Update $U$ is an execution by $P_i$?
2. The lemma says “To prove the lemma it suffices etc.” Why this is true?
3. Show in details how $\omega(r_1) = \omega(r_2)$ is concluded at the end of the lemma.
For any indirect Scan operation $S$ we define $Report(S)$ as follows. Let $U$ be that Update operation that Lemma 2.7 provides, and define $Report(S) = Scan(U)$. So

$$val(S) = val(Report(S)) \text{ and } Report(S) \sqsubseteq S.$$  \hfill (2)

For any Scan operation $S_0$ we define the report sequence of $S_0$, to be the sequence $S_0 \sqsubseteq S_1 \sqsubseteq \cdots \sqsubseteq S_N$ obtained by the following inductive definition. Suppose that $S_n$ is defined. If $S_n$ is a direct Scan execution then we define $S_{n+1} = S_n$. If $S_n$ is an indirect Scan operation then we define $S_{n+1} = Report(S_n)$. Since $S \not\sqsubseteq T$ implies that $S$ and $T$ belong to different processes, there is some $n \leq N$ such that $S_0 \sqsubseteq \cdots \sqsubseteq S_n = S_{n+1}$ (and $S_n = S_k$ for all $k > n$). That is, $S_0, \ldots, S_{n-1}$ are all indirect Scan operations but $S_n$ is a direct one. Thus $\Omega(S_n)$ is defined, and we define $\Omega(S_0) = \Omega(S_n)$. We also define $\Omega_i(S_0) = \Omega_i(S_n)$ for $1 \leq i \leq N$.

**Theorem 2.8** The following properties of the function $\Omega_i$ hold (for $1 \leq i \leq N$). For any Scan operation $S$ (direct or indirect):

1. $W_i = \Omega_i(S)$ is an Update operation by $P_i$, and $\text{end}(W_i) < \text{end}(S)$.

   $$val(S) = (val(W_1), \ldots, val(W_N)).$$

2. If $V$ is any Update operation by $P_i$ such that $V < S$ then $V \leq \Omega_i(S)$.

3. If $V$ is an Update operation by $P_j$ such that $\text{end}(V) < \text{end}(\Omega_i(S))$ ($i$ is any index $1 \leq i \leq N$) then $V \leq \Omega_j(S)$.

**Proof.** Consider the report sequence $S = S_0 \sqsubseteq \cdots \sqsubseteq S_n$ where $n$ is the first index such that $S_n = S_{n+1}$. Then

$$val(S_0) = val(S_n)$$  \hfill (3)

follows from (2)) and likewise $S_n \sqsubseteq S_0$. So the properties that were established in Lemma 2.6 transfer to the present theorem. For example, the argument for item 2 is the following. If $V$ is an Update operation by $P_i$ such that $V < S_0$, then $V < S_n$ (follows from $S_n \sqsubseteq S_0$) and hence $V \leq \Omega_i(S)$ by Lemma 2.6.

**Exercise 6** Prove in details (3), and prove item 3 of the lemma.
We are now ready for the third phase of the proof: showing that the higher-level properties of the Ω function established in Theorem 2.8 imply the snapshot properties. To make it absolutely clear that the statement of this third phase theorem and its proof are independent of the algorithm, we first restate the properties established in Theorem 2.8 in a separate list in Figure 2. There is an important difference between the formulation in Theorem 2.8 and the properties listed in Figure 2. We have mentioned that there are two kinds of Scan operations: standalone operations, and Scan operations that are part of some Update operation. So far there was no need to separate these operations since they have the same properties, and what theorem 2.8 says is true both for the standalone operations and the Scan operations that are invoked by Update operations. In the list of abstract properties of Figure 2 the term “Scan events” refer to the standalone Scan events, and there is no need to refer to those Scan events that are part of the Update events. The point is that now we are interested in the Update/Scan operations, and the fact that the Update operations need (for the algorithm to work) to invoke Scan operations is a detail that is abstracted away at this stage of the proof.

The list of properties of Figure 2 is written in such a language that it can also be applied to the multiwriter algorithm. In first reading this list and the proof of Theorem 2.9 the reader can assume that \( M = N \).

The system-execution language in which the properties of Figure 2 are stated is a moment-based system execution language that has the following components. There are two sorts: the Event and Data sorts. There are unary predicates Scan and Update and \( P_1, \ldots, P_N \) on the Event sort. The binary precedence relation symbol \(<\). We have the following function symbols. The \( \text{val} \) function returns Data values when defined on Update events, and on Scan events \( \text{val} \) returns \( M \)-tuples of Data values. We have \( \Omega_x : \text{Event} \rightarrow \text{Event} \) for the function that returns the \( 1 \leq x \leq M \)-th component of \( \Omega \). That is, for every Scan event \( S \), \( \Omega(S) = (\Omega_1(S), \ldots, \Omega_M(S)) \). We also have a function that give the address of every Update operation. That is, if \( U \) is an Update operation and \( \text{address}(U) = x \) then \( U \) is said to be an update on address \( A_x \). We also have functions \( \text{begin} \) and \( \text{end} \) defined on the Event sort. Intuitively, \( \text{begin}(E) \) and \( \text{end}(E) \) denote the start and end of event \( E \) in the Moment sort. There are two properties of the \( \text{end} \) function which we shall use.

\( \text{E1} \) If \( X < Y \) are Scan/Update events and \( X < Y \) then \( \text{end}(X) < \text{end}(Y) \).

\( \text{E2} \) For every events \( A \) and \( B \), if \( A \neq B \) then either \( \text{end}(A) < \text{end}(B) \) or
S1 For every address index $1 \leq x \leq M$, if $U_1$ and $U_2$ are two different Update events on $A_x$ (that is $\text{address}(U_1) = \text{address}(U_2) = x$) then $U_1 < U_2$ or else $U_2 < U_1$.

S2 For every Scan event $S$ and for every memory address $A_x$ (so $1 \leq x \leq M$), $W_x = \Omega_x(S)$ is an Update event on $A_x$ by some process, and $\text{end}(W_x) < \text{end}(S)$.

$$\text{val}(S) = (\text{val}(W_1), \ldots, \text{val}(W_M)).$$

S3 If $V$ is any Update operation on address $A_x$ and $S$ is any Scan event such that $V < S$, then $V \leq \Omega_x(S)$.

S4 If $V$ is an Update operation on address $A_y$ and $S$ is a Scan event such that $\text{end}(V) < \text{end}(\Omega_x(S))$ ($x$ is any address index $1 \leq x \leq M$), then $V \leq \Omega_y(S)$.

Figure 2: The higher level properties of the snapshot algorithm.

else $\text{end}(B) < \text{end}(A)$.

**Exercise 7** Write the properties of Figure 2 in the formal first-order language described above.

**Theorem 2.9** Let $M$ be any system execution that satisfies the properties listed in Figure 2. Then there is a linear ordering on the Scan/Update events that satisfy the linear snapshot specifications of Section 1.

The proof begins with the following lemma.

**Lemma 2.10** Let $S_1$ and $S_2$ be two Scan events and assume that for some $1 \leq x \leq M$ $\Omega_x(S_1) < \Omega_x(S_2)$. Then for every $1 \leq y \leq M$ $\Omega_y(S_1) \leq \Omega_y(S_2)$.

**Proof.** Suppose on the contrary that $\Omega_x(S_1) < \Omega_x(S_2)$, but for some $1 \leq y \leq N$

$$\neg(\Omega_x(S_1) < \Omega_x(S_2)). \tag{4}$$

Then the following is deduced.

11
1. $\Omega_y(S_2) < \Omega_y(S_1)$.
2. Either $\text{end}(\Omega_y(S_1)) < \text{end}(\Omega_x(S_2))$ or $\text{end}(\Omega_x(S_2)) < \text{end}(\Omega_y(S_1))$.
3. Suppose first that $\text{end}(\Omega_y(S_1)) < \text{end}(\Omega_x(S_2))$. Then $\Omega_y(S_1) \leq \Omega_y(S_2)$.
4. This is in contradiction to $\Omega_y(S_2) < \Omega_y(S_1)$.
5. Now assume that $\text{end}(\Omega_x(S_2)) < \text{end}(\Omega_y(S_1))$. Then $\Omega_x(S_2) \leq \Omega_x(S_1)$.
6. This is in contradiction to $\Omega_x(S_1) < \Omega_x(S_2)$.
7. Thus in both cases we have reached a contradiction and hence the lemma follows.

Exercise 8 Justify each of the items above.

Let $X$ and $Y$ be two (different) Scan/Update events. We define $X <^* Y$ if and only if one of the following items 1 – 4 holds:

O1 $X$ and $Y$ are both Update events and $\text{end}(X) < \text{end}(Y)$.

O2 $X$ is an Update event on address $A_y$, $Y$ a Scan event, and $X \leq \Omega_y(Y)$.

O3 $X$ is a Scan event, $Y$ an Update event on address $A_y$, and $\Omega_y(X) < Y$.

O4 $X$ and $Y$ are both Scan events and one of the following possibilities is the case:

(a) For some $1 \leq y \leq M$, $\Omega_y(X) < \Omega_y(Y)$, or
(b) For all $1 \leq y \leq M$ $\Omega_y(X) = \Omega_y(Y)$ and $\text{end}(X) < \text{end}(Y)$.

We claim that $<^*$ is the required linear ordering that proves that the algorithm implements an atomic Scan/Update system. We have to prove the following.

Lemma 2.11 1. Relation $<^*$ extends the precedence relation $<$.

2. $<^*$ is irreflexive.

3. For any two Scan/Update events $X \neq Y$, $X <^* Y$ or $Y <^* X$.

4. Relation $<^*$ is a transitive relation on the Scan/Update events.
5. For any Scan event $S$, if $\text{val}(S) = (d_1, \ldots, d_M)$ then, for every $1 \leq y \leq M$, $d_y = \text{val}(W_y)$ where $W_y$ is the $<^*\text{-last Update event}$ $W$ on address $A_y$ such that $W <^* S$.

**Exercise 9** Prove these five items of the lemma using the following guidelines.

1. Assume that $X < Y$ are $\text{Scan}/\text{Update}$ events. Prove (in great details) that $X <^* Y$ for each of the following cases.
   
   (a) $X$ and $Y$ are both $\text{Update}$ events.
   
   (b) $X$ is an $\text{Update}$ event, $Y$ is a $\text{Scan}$ event.
   
   (c) $X$ is a $\text{Scan}$ event and $Y$ is an $\text{Update}$ event.
   
   (d) $X$ and $Y$ are $\text{Scan}$ events.

2. This is trivial since we define $X <^* Y$ only when $X \neq Y$, and so $X <^* X$ is impossible.

3. Suppose that $X$ and $Y$ are two different $\text{Scan}/\text{Update}$ events. Prove that $X <^* Y$ or $Y <^* X$ in each of the following cases.
   
   (a) $X$ and $Y$ are both $\text{Update}$ events.
   
   (b) $X$ and $Y$ are both $\text{Scan}$ events.
   
   (c) One of $X$ and $Y$ is a $\text{Scan}$ and the other an $\text{Update}$ event. Say $X$ is a $\text{Scan}$ and $Y$ an $\text{Update}$ event.

4. Assume that $X <^* Y <^* Z$. Prove that $X <^* Z$ in each of the following cases.
   
   (a) $X$ and $Y$ are both $\text{Update}$ events.
      
      i. $Z$ is an $\text{Update}$ event.
      
      ii. $Z$ is a $\text{Scan}$ event.
   
   (b) $X$ is an $\text{Update}$ event, $Y$ is a $\text{Scan}$ event.
      
      i. $Z$ is an $\text{Update}$ event.
      
      ii. $Z$ is a $\text{Scan}$ event.
   
   (c) $X$ is a $\text{Scan}$ event and $Y$ is an $\text{Update}$ event.
i. $Z$ is an Update event.
ii. $Z$ is a Scan event.

(d) $X$ and $Y$ are Scan events. Here you can use Lemma 2.10.

i. $Z$ is an Update event.
ii. $Z$ is a Scan event.

5. Finally, prove that $<^*$ is correct. (This is immediate, but nevertheless give the details.)

**Discussion.** It may seem quite strange that the correctness proof that we gave here to the snapshot algorithm did not consider the notion of state. One may rightly argue that in order to prove the correctness of any algorithm it is necessary to explicate the connection between the text of the algorithm and the resulting executions, and since states are necessary for such an explication no correctness proof can do without them. Indeed states are necessary, but my claim here is that only local states are necessary—and sufficient. We can do without global states, and in certain cases it makes sense to avoid them. (A local state of process $P$ is a function defined on the local variables of $P$ which assigns to each such variable a value in its type. A global state gives, in addition to the local states of all processes, information on the states of the communication devices.)

In order to prove Lemma 2.2, for example, we have to consider the algorithm of the Scan operation in isolation. The local state variables of the Scan operation are the variables $k$, $j$, $PC$ (for the program counter), and we also have $N^2 + 2N$ variables $a_k[j]$ for $0 \leq k \leq N + 1$ and $1 \leq j \leq N$. The values of these $a_k[j]$ variables are register triples with fields $(data, sequence-number, report)$ where $data$ is in $Data$, $sequence-number$ is a natural number, and $report$ is an array of length $N$ of $Data$ values (see section 2). Registers $R_i$ that the Scan operation reads are not among its local state variables. This is a characteristics feature of the unrestricted semantics that we give here to the standalone Scan semantics. A way to think about it is by realizing that the correctness of Lemma 2.2 can be proved even when the values returned by reads of registers $R_j$ are determined randomly (but within their types). The resulting semantics is said to be “unrestricted” in [1]. In the unrestricted semantics, each process executes its algorithm but the registers that are read return arbitrary values (and the write actions are not recorded in register variables since the registers are not state variables).
After establishing the unrestricted semantics with local states, the restricted semantics is obtained by imposing on the unrestricted semantics the specifications of the registers (atomicity or seriality), and the correctness proof thus relies on the combination of the unrestricted semantics and the specifications of the communication devices.

3 Multiwriter snapshot

The snapshot algorithm that we have described in the previous sections is a single-writer snapshot algorithm in which process $P_i$ is the sole writer on address $A_i$, and now we want to describe a multiwriter snapshot. In the multiwriter case, memory addresses are no longer associated with a specific process but rather any process can write on any memory address. Thus, if the multiwriter memory addresses are $A_1, \ldots, A_M$ then an $Update$ invocation has the form $Update(x, v)$ where $1 \leq x \leq M$ refers to the memory address $A_x$ on which the value $v \in Data$ is to be written. This operation can be executed by any of the processes.

The algorithm that we describe here in Figure 3 is due to D. Imbs and M. Raynal [4]. The processes are $P_1, \ldots, P_N$, and each process $P_i$ has a single writer regular register $HLP_i$ whose values are sequences $(d_1, \ldots, d_M)$ of $Data$ values. Then there are multiwriter serial registers $MW_1, \ldots, MW_M$ that any of these processes can use. If $a = (v, i)$ is the value of some register $MW_i$ then we write $v = a.data$ and $i = a.id$ for the two fields of $a$ ($1 \leq i \leq N$). We assume that process $P_i$ never write twice the same value $v \in Data$. A simple way to obtain this is by asking $P_i$ to keep a counter $seq_i$ that is increased by 1 at every $Update$ execution, and to incorporate $seq_i$ into the value $v$ that is written.

A variant of the multiwriter algorithm of [4] is in Figure 3. Here too, we present a less efficient algorithm in the sense that the $Scan$ operation always executes $N + 2 \ collect()$ operations, whereas the original algorithm of [4] can return immediately after two successive collects that return identical values. We have chosen this form in order to have a somewhat simpler presentation in class. The reader can easily modify the algorithm (after understanding its principle) to the more efficient form.

An important idea of the algorithm of [4] is the “write-first-help-later” approach which separates the values of an $Update$ operation into two registers. In an execution of $Update(x, v)$ by $P_i$ the value $(v, i)$ is first written on the
Update \((x, v)\) by \(P_i\): (* write of \(v \in Data\) on memory address \(A_x\). *)

0. \(MW_x := (v, i)\);
1. \(s := Scan()\);
2. \(HLP_i := s\)

Scan (by any process)

0. \(\text{for } k := 0 \text{ to } N + 1 \ a_k := \text{collect()}\);
1. \(\text{Case 1: for some } k \in \{0 \cdots N\} \ a_k = a_{k+1} \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \text{return } (a_k[1].data, \ldots, a_k[N].data)\);
2. \(\text{Case 2: not Case 1. For some } 1 \leq i \leq N \text{ there are } m, n \text{ such that } 0 \leq m < n < N + 1 \text{ and there are register indexes } 1 \leq x, y \leq M \text{ such that } (a_m[x] \neq a_{m+1}[x]) \land (a_n[y] \neq a_{n+1}[y]) \land (a_{m+1}[x].id = a_{n+1}[y].id = i)\)
3. \(h := HLP_i\)
   \(\text{return } h\)

collect():

0. \(\text{for every } 1 \leq j \leq M \ a[j] := MW_j;\)
1. \(\text{return } a;\)

Figure 3: A variant of the multiple-writer snapshot algorithm of [4].

multiwriter register \(MW_x\), and then the array \(s\) returned from a Scan operation is written on the singlewriter register \(HLP_i\) of the executing process \(P_i\). This write action on \(MW_x\) is in fact the “linearization point” of the Update operation that contains it. For this reason and in order to highlight the similarities between the single-writer and multi-writer snapshot algorithms, we say that a full Update operation is an execution of lines 0,1,2 of the Update code of Figure 3, but we say that \(U\) is an Update operation on address \(A_x\) when \(U\) is the write action on register \(MW_x\), that is the execution of line 0 in some full Update operation. Thus an Update operation \(U\) consists of a single action, and hence \(end(U) = U\).

The Scan algorithm is a little bit more complicated than the corresponding single writer snapshot algorithm, but not much more. The first case, Case 1, is the same here as in the algorithm of [2] that we presented earlier. The second case is somewhat different. The point is that we no longer have \(N\) registers as in the single writer case. The number \(M\) of registers can be
much larger than the number of processes $N$, and so the pigeon hole principle can no longer yield a single multiwriter register that has changed twice its value. Fortunately it’s not the same register but the same process that has to change its values. This is seen in the following lemma which shows that if the first case does not apply then necessarily Case 2 applies.

**Lemma 3.1** For any Scan execution $S$, if Case 1 does not apply then Case 2 holds.

**Exercise 10** Prove the lemma.

### 3.1 Correctness of the multiwriter Snapshot algorithm.

We are going to prove in this section that executions of this multiwriter snapshot algorithm satisfy the same higher-level properties displayed in Figure 2 that executions of the single writer algorithm satisfy. This not only yields a shorter correctness proof, but will also show the affinity between the two algorithms (which is evident on the intuitive level but is not so obvious to express formally). Not only that, but we will see that the registers $HLP_i$ on which the $Update$ operations by $P_i$ write their reports are not required to be atomic–regular registers work as well. This is interesting because it is very difficult to prove the correctness of algorithms that use regular registers if the proof is carried within the global states and histories framework. Since we are not using global states in our proof, we have no accrued difficulty in dealing with regular registers. The proof that we give here is not much different from the proof for the earlier algorithm that employed only atomic registers.

Corresponding to lemmas 2.2 and 2.3 we have here the following two lemmas which form the first phase of the correctness proof, that is the phase of the unrestricted semantics. Lemma 3.2 is about executions of the Scan algorithm with arbitrary behavior of the registers, and Lemma 3.3 is about executions of the $Update(x, v)$ algorithm again with no restriction on the registers’ behavior.

**Lemma 3.2** Any Scan operation $S$ is either direct or indirect (that is, in Case 1 or in Case 2).

1. If $S$ is direct then there are two collect executions $c_1 < c_2$ in $S$ such that $val(c_1) = val(c_2)$. Say $a = val(c_1)$ is this common array. The return value of $S$ is $val(S) = (a[1].data, \ldots, a[M].data)$. 

17
2. If $S$ is indirect then there is some index $1 \leq i \leq N$ and there are register indexes $1 \leq x, y \leq M$ such that there are four read actions in $S$, $r_1 < r_2 < r_3 < r_4$, such that $r_1$ and $r_2$ are reads of register $MW_x$, $val(r_1) \neq val(r_2)$, and $r_3, r_4$ are reads of register $MW_y$, $val(r_3) \neq val(r_4)$ and $val(r_2).id = val(r_4).id = i$. Moreover, there is in $S$ a read $r$ of the regular register $HLP_i$, such that $r_4 < r$ and the value of $S$ in this case is $val(S) = val(r)$.

**Lemma 3.3** If $U$ is a full Update operation by process $P_i$ on the register with index $1 \leq x \leq M$, and if $val(U) = v$ is the value of $U$, then $U$ consists of a write action on register $MW_x$ of the value $(v, i)$ which is followed by a Scan execution $S = Scan(U)$, which is followed by a write action on register $HLP_i$ of the value $val(S)$.

Note: a full Update operation is an execution of lines 0, 1, 2 of the Update code, and if we say that $U$ is an Update operation then we mean by that $U$ is the write on the $MW_x$ register. So, $U$ and $\text{end}(U)$ both denote the write action on the multiwriter register.

The second phase of the proof combines the statements of lemmas 3.2 and 3.3 with the semantics of serial and regular registers. As a result of our identification of Update operations on address $A_x$ with the write actions on register $MW_x$, seriality of this register implies immediately that if $U_1$ and $U_2$ are two Update operations on address $A_x$ then either $U_1 < U_2$ or else $U_2 < U_1$.

If $r$ is any read action of register $MW_x$ then $\omega(r)$ denotes the last write action $w$ on that register such that $w < r$. Following the semantics of serial registers we get that $val(\omega(r)) = val(r)$. And if $r$ is a read of a regular register $HLP_i$ then $\omega(r)$ is a write on that register such that $val(\omega(r)) = val(r)$ and the following two properties hold. One, it is not the case that $r < \omega(r)$ and two, there is no write action $w$ on that register such that $\omega(r) < w < r$.

**Definition 3.4** Let $S$ be a direct Scan operation. We define $\Omega(S) = (\Omega_1(S), \ldots, \Omega_M(S))$ as follows.

By Lemma 3.2 there are two collect executions $c_1 < c_2$ in $S$ such that $val(c_1) = val(c_2)$, and if $a = val(c_1)$ then $val(S) = (a[1].data, \ldots, a[M].data)$. For every index $1 \leq x \leq M$ we let $r(x) \in c_1$ be the read action of register $MW_x$ in $c_1$. Then we define $\Omega_x(S) = \omega(r(i))$ as the corresponding write action on $MW_x$ (which is, by our definition, an Update operation on $A_x$).
Lemma 3.5 The following properties of the function $\Omega_x$ hold (for $1 \leq x \leq M$). For any direct Scan operation $S$:

1. $W_x = \Omega_x(S)$ is an Update operation by some $P_i$ on address $A_x$, and $\text{end}(W_x) < \text{end}(S)$. (As we have said, $W_x = \text{end}(W_x)$ is the write action on register $MW_x$.)

   $$\text{val}(S) = (\text{val}(W_1).data, \ldots, \text{val}(W_M).data).$$

2. If $V$ is any Update operation on address $A_x$ such that $V < S$ then $V \leq \Omega_x(S)$.

3. If $V$ is an Update operation on address $A_y$ such that $\text{end}(V) < \text{end}(\Omega_x(S))$ then $V \leq \Omega_y(S)$.

Note that an Update operation on address $A_x$ is (by our definition) just a write action on the serial multi-writer register $MW_x$.

Exercise 11 Prove the lemma.

The analysis of indirect Scan operations is somewhat different from the corresponding analysis of the previous algorithm because the return of the Scan depends now on the read of the regular register $HLP_i$.

Lemma 3.6 Suppose that $S$ is an indirect Scan operation. Then there is a full Update operation $U$ such that the following holds for $P = \text{Scan}(U)$.

1. $\text{val}(S) = \text{val}(P)$.

2. $P \sqsubseteq S$.

Exercise 12 Prove this lemma.

Note, this is possibly a longer proof than the other exercises.

For any indirect Scan operation $S$ we define $\text{Report}(S)$ as as before, and equation (2) holds for the multiwriter algorithm as well.

$$\text{val}(S) = \text{val}(\text{Report}(S)) \text{ and } \text{Report}(S) \sqsubseteq S. \quad (5)$$

For any Scan operation $S_0$ we define the report sequence of $S_0$ as before: it is the sequence $S_0 \sqsubseteq S_1 \sqsubseteq \cdots$ obtained by the rule that $S_{n+1} = \text{Report}(S_n)$ if $S_n$ is an indirect Scan operation. Since $S \sqsupseteq T$ implies that $S$ and $T$ belong to different processes, there is some $n \leq N$ such that $S_0, \ldots S_{n-1}$ are all indirect Scan operations but $S_n$ is a direct one. Thus $\Omega(S_n)$ is defined, and we define (as before) $\Omega(S_0) = \Omega(S_n)$ and $\Omega_x(S_0) = \Omega_x(S_n)$ for $1 \leq x \leq M$.
Exercise 13  Check that all the higher-level properties of Figure hold for the multiwriter algorithm as well.

References


