Chapter 1

Atomicity: A Correctness Property for Shared Objects

October 1, 2007

1 Introduction

Before diving into how to implement shared objects, we first address in this chapter more elementary questions on the very meaning of a shared object and its correctness. In particular, we ask:

- How to specify the behavior of a shared object?
- What is a correct execution of a set of processes accessing a set of shared objects? In particular, how to measure correctness if one or more processes stop their execution in the middle of an operation?

To give a flavor of the questions we address, let us consider an unbounded FIFO (first in first out) queue. This is an object of the type queue defined by the following two operations:

- $Enq(v)$: Add the value v at the end of the queue,
- $Deq()$: Return the first value of the queue and suppress it from the queue; if the queue is empty, return the default value $\perp$.

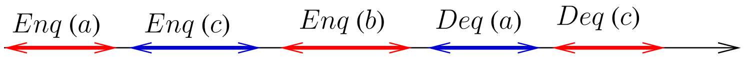

Figure 1: A sequential execution

Figure 1 describes a sequential execution of a system made up of a single process using the queue. The time-line, going from left to right, describes the progress of the process when it enqueues first the value a, then the value c, and finally the value b. According to the expected semantics of a queue, and as depicted by the figure, the first invocation of $Deq()$ returns the value a, the second returns the value c, etc.

Figure 2 depicts an execution of a system made up of two processes sharing the same the queue. Now, process p_1 enqueues a and then b whereas process p_2 concurrently enqueues c. On the figure, the execution of $Enq(c)$ by p_2 overlaps both $Enq(a)$ and $Enq(b)$ by p_1. Such execution raises the following questions:

- What values are dequeued by p_1 and p_2?

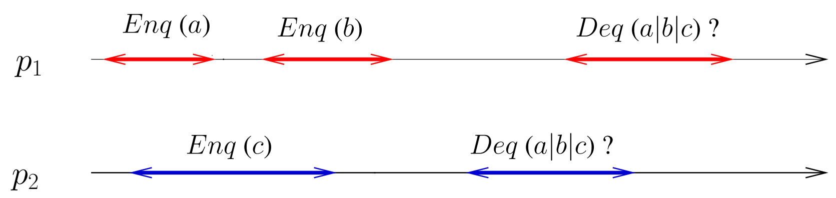

Figure 2: A concurrent execution

- What values can be returned by a process if the other process has failed while executing an operation?
- What happens if p_1 and p_2 share several queues instead of a single one? Etc.

Addressing these and related questions goes first through defining more precisely our model of computation.

2 Model

2.1 Processes and operations

The system we consider consists of a finite set of n *processes*, denoted $p_1, \ldots, p_n$. The processes execute some common distributed computation and, while doing so, cooperate by accessing *shared objects*.

Processes synchronize their activities by executing operations exported by shared objects. An execution by a process of an operation on an object X is denoted $X.op(arg)(res)$ where arg and res denote respectively the input and output parameters of the invocation. The output corresponds to the response to the invocation. Sometimes we simply write $X.op$ when the input and output parameters are not important. The execution of an operation $op()$ on an object X by a process p_i is modeled by two events, namely, the events denoted $inv[X.op(arg)$ by $p_i]$ that occurs when p_i invokes the operation (invocation event), and the event denoted $resp[X.op(res)$ by $p_i]$ that occurs when the operation terminates. (When there is no ambiguity, we talk about *operations* where we should be talking about *operation executions*.) We say that these events are generated by the process p_i and associated with the object X. Given an operation $X.op(arg)(res)$, the event $resp[X.op(res)$ by $p_i]$ is called the response event matching the invocation event $inv[X.op(arg$ by $p_i]$.

An execution of a distributed system induces a sequence of interactions between the processes of the system and the shared objects. Every such interaction corresponds to a computation *step* and is represented by an *event*: the visible part of a step, i.e., the invocation or the reply of an operation. A sequence of events is called a *history* and this is precisely how we model executions. We will detail this later in this chapter.

As we pointed out in the introduction of the book, we generally assume that processes are *sequential*: a process executes (at most) one operation of an object at a time. That is, the algorithm of a sequential process stipulates that after an operation is invoked on an object and until a matching response is received, the process does not invoke any other operation. The fact that processes are individually sequential does not preclude them from concurrently invoking operations on the same shared object. Sometimes, we will focus on *sequential executions* (modeled by *sequential histories*) which precisely preclude such concurrency; that is, only one process at a time invokes an operation on an object in a sequential execution.

2.2 Objects

An object has a name and a type. A type is defined by (1) the set of possible values for (the states of) objects of that type; (2) a finite set of operations through which the objects of that type can be manipulated; and (3) a specification describing, for each operation, the condition under which that operation can be invoked, and the effect produced after it has been executed. Figure 3 presents a structural view of a set of n processes sharing m objects.

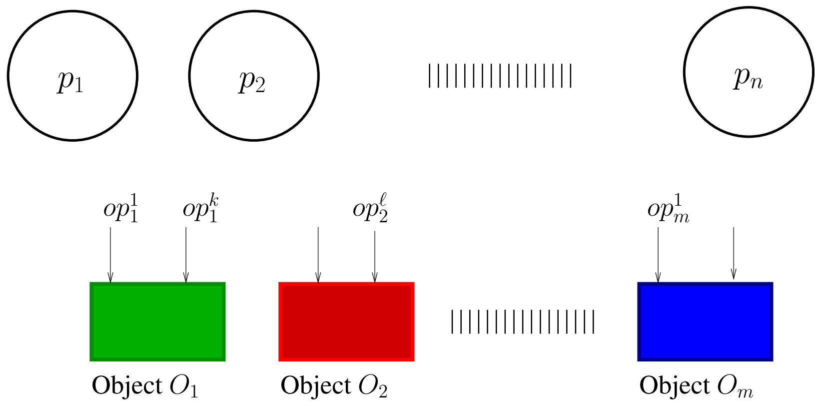

Figure 3: Structural view of a system

Sequential specification The object types we consider do generally have a sequential specification. We talk both about the specification of the object or the specification of the type. A sequential specification depicts the behavior of the object when accessed sequentially, i.e., in a sequential execution.

One can describe a sequential specification by associating two predicates with each operation. These predicates are called pre-assertion and post-assertion. Assuming the pre-assertion is satisfied before executing the operation, the post-assertion describes the new value of the object and the result of the operation returned to the calling process. We say that an object operation is *total* if it is defined for every state of the object; otherwise it is *partial*. This means that, differently from the pre-assertion associated with a partial operation, the pre-assertion associated with a total operation is always satisfied.

We also say that an object operation is *deterministic* if, given any state of the object that satisfies the pre-assertion and input parameters, the output parameters and the final state of the object are uniquely defined. An object (resp., type) that has only deterministic operations is said to be deterministic; otherwise we say that the object (resp., type) is non-deterministic.

Example 1: a read/write object (register) To illustrate the notion of sequential specification, we consider here three examples object types. The first type (called the type `register`) is a simple read/write abstraction, that models objects such as a shared memory word, a shared file or a shared disk.

It has two operations:

- The operation $read()$ has no input parameter. It returns a value of the object.
- The operation $write(v)$ has an input parameter, v, a new value of the object. The result of that operation is a value ok indicating to the calling process that the operation has terminated.

The sequential specification of the object is defined by all the sequences of read and write operations in which each read operation returns the value of the last preceding write operation (i.e., the last value written). Clearly, the read and write operations are always defined: they are total operations.

The problem of implementing a concurrent read/write object is a classical synchronization problem known under the name *reader/writer* problem.

Example 2: a FIFO queue with total operations The second example is the unbounded (FIFO) queue described in Section 1. Such object has the following sequential specification: every dequeue returns the first element enqueued and not dequeued yet. If there is not such element (i.e., the queue is empty), a specific default value $\perp$ is returned. This definition never prevents an enqueue or a dequeue operation to be executed. Both enqueue and dequeue operations are total.

Example 3: a FIFO queue with a partial operation Let us consider now the previous queue definition modified as follows: a dequeue operation can be executed only when the queue is not empty. The sequential specification of this object is then a restriction of the previous specification; all situations where a dequeue operations returns $\perp$ have to be precluded. The enqueue operation can always be executed, so it remains a total operation. On the other hand, the pre-assertion of the dequeue operation states that it can only be executed when the queue is not empty; consequently, that operation is a partial operation.

The two FIFO queues examples (2 and 3) are two variants of a the classical *producer/consumer* synchronization problem.

Example 4: an object with no sequential specification Not all object types have a sequential specification. To illustrate this, let us consider a *rendezvous* object that can be accessed by two processes p_1 and p_2. Such an object provides the processes with a single operation $meeting()$ with the following semantics: after it has been invoked by a process, the operation terminates only when the other process has also invoked the operation. In other words, the key property of this object is that no process can terminate an operation without a concurrent invocation. It is easy to see that such a rendezvous object has no sequential specification: the behavior of the object cannot be described simply by stating what happens when the operation invocations by p_1 and p_2 would be totally ordered. (A rendezvous object is a typical example of an object that has no sequential specification. In this book, we are mainly interested in objects that have a sequential specification.)

2.3 Histories

The notion of execution of a set of processes accessing a set of shared objects is captured by the concept of *history*.

Representing an execution as a history of events If we assume that simultaneous (invocation and response) events are independent, which is generally the case, in particular for events generated by sequential processes accessing objects with a sequential specification, it is always possible, without loss of generality, to arbitrarily order simultaneous events. This makes it possible to consider a total order relation on the events (denoted $<_H$ in the following) abstracting the real-time order in which the events do actually occur.

The interaction between a set of sequential processes and a set of shared objects is modeled by a sequence of invocation and response events, called a *history* (sometimes also called a *trace*), and denoted

$\widehat{H} = (H, <_H)$ where H is the set of events generated by the processes and $<_H$ a total order on these events. The objects and processes associated with events of $\widehat{H} = (H, <_H)$ are said to be involved in $\widehat{H}$. $\widehat{H}|p_i$ ($\widehat{H}$ at p_i) is called a *local* history; it denotes the subsequence of $\widehat{H}$ made up of all the events generated by the process p_i.

Complete vs partial histories An operation is said to be *complete* in a history if the history includes both the event corresponding to the invocation of the operation and its response. Otherwise we say that the operation is *pending*. A history without pending operations is said to be *complete*. A history with pending operations is said to be *partial*. Note that, being sequential, a process can have at most one pending operation in a given history.

Equivalent histories Two histories $\widehat{H}$ and $\widehat{H'}$ are said to be *equivalent* if they have the same local histories, i.e., for each p_i, $\widehat{H}|p_i = \widehat{H'}|p_i$. That is, equivalent histories are built from the same set of events (remember that an event includes the name of an object, the name of a process, the name of an operation and input -or output- parameters).

Well-formed histories As we are interested only in histories generated by sequential processes, we restrict our attention to the histories $\widehat{H}$ such that, for each process p_i, $\widehat{H}|p_i$ (the local history generated by p_i) is sequential: it starts with an invocation, followed by a response, called the matching response and associated with the same object, followed by another invocation, etc. We say in this case that $\widehat{H}$ is *well-formed*.

Partial order on operations A history $\widehat{H}$ induces an irreflexive partial order on its operations as follows. Let $op = X.op1()$ by p_i and $op' = Y.op2()$ by p_j be two operations. Informally, operation op precedes operation op', if op terminates before op' starts, where "terminates" and "starts" refer to the time-line abstracted by the $<_H$ total order relation. More formally:

$$\big(op \rightarrow_H op'\big) \stackrel{\text{def}}{=} \big(resp[op] <_h inv[op']\big).$$

Two operations op and op' are said to *overlap* (we also say are *concurrent*) in a history $\widehat{H}$ if neither $resp[op] <_H inv[op']$, nor $resp[op'] <_H inv[op]$. Notice that two overlapping operations are such that $\neg(op \rightarrow_H op')$ and $\neg(op' \rightarrow_H op)$. As a sequential history has no overlapping operations, it follows that $\rightarrow_H$ is a total order if $\widehat{H}$ is a sequential history.

Illustrating histories Figure 4 depicts a well-formed history $\widehat{H}$. The history comprises ten events $e1 \ldots e10$ ($e4,\ e6,\ e7$ and $e9$ are explicitly detailed). As there is a single object, its name is omitted. It is easy to see that the enqueue operation issued by p_2 overlaps both enqueue operations issued by p_1. Notice that the operation $Enq(c)$ by p_2 is concurrent with both $Enq(a)$ and $Enq(b)$ issued by p_1. Moreover, the history $\widehat{H}$ has no pending operations, and is consequently complete.

To illustrate the notions of partial and complete histories, let us again consider Figure 4. The sequence $e1 \ldots e9$ is a partial history where the dequeue operation issued by p_1 is pending. The sequence $e1 \ldots e6\ e7\ e8\ e10$ is another partial history in which the dequeue operation issued by p_2 is pending. Finally, the history $e1 \ldots\ e8$ has two pending operations. We are now ready to define more precisely sequential histories.

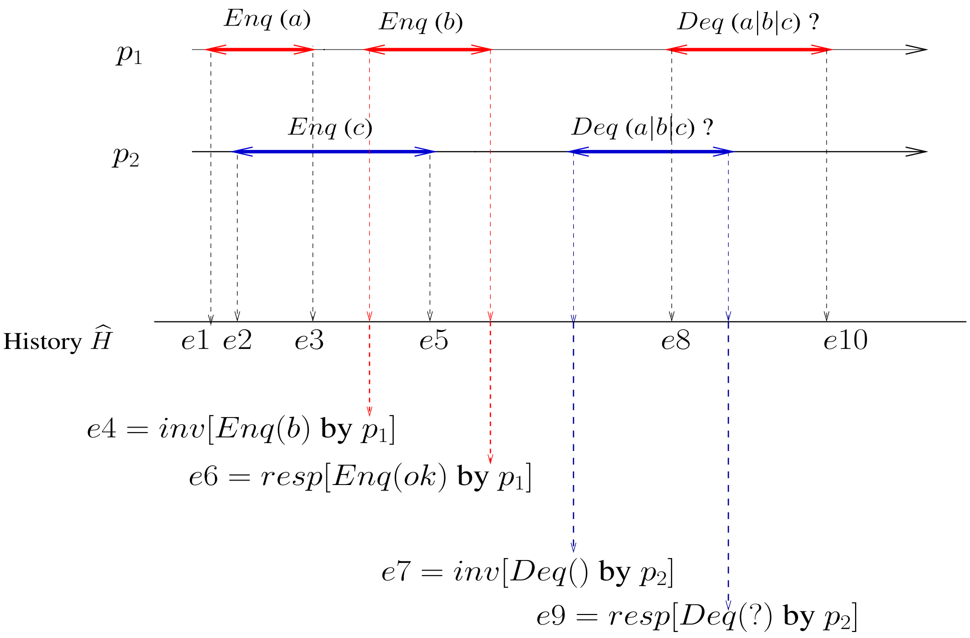

Figure 4: Example of a history

2.4 Sequential history

Definition In some cases, it is easier to reason on executions at the granularity level defined by the operations issued by the processes, and not at the granularity level defined by the events they generate. The notion of sequential history is introduced to that end.

A history is *sequential* if its first event is an invocation, and then (1) each invocation event, except possibly the last, is immediately followed by the matching response event, and (2) each response event, except possibly the last, is immediately followed by an invocation event. The sentence "except possibly the last" associated with an invocation event is due to the the fact that a history can be partial. A complete sequential history always ends with a response event. A history that is not sequential is *concurrent*.

A sequential history models a sequential multiprocess computation (there are no overlapping operations in such a computation), while a concurrent history models a concurrent multiprocess computation (there are at least two overlapping operations in such a computation). Given that a sequential history $\widehat{S}$ has no overlapping operations, the associated partial order $\rightarrow_S$ defined on its operations is actually a total order.

Strictly speaking, the sequential specification of an object is a set of sequential histories involving solely that object. Basically, the sequential specification represents all possible sequential accesses to the object.

Example Considering Figure 4, $\widehat{H}$ is a complete concurrent history. On the other hand, the complete history

$$\widehat{H_1} = e1\ e3\ e4\ e6\ e2\ e5\ e7\ e9\ e8\ e10$$

is sequential: it has no overlapping operations. We can thus highlight its sequential nature by separating its operations using square brackets as follows:

$$\widehat{H_1} = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e7\ e9]\ [e8\ e10].$$

The following histories $\widehat{H_2}$ and $\widehat{H_3}$

$$\widehat{H_2} = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e8\ e10]\ [e7\ e9],$$

$$\widehat{H_3} = [e1\ e3]\ [e4\ e6]\ [e8\ e10]\ [e2\ e5]\ [e7\ e9].$$

are also sequential. Let us also notice that $\widehat{H}$, $\widehat{H_1}$, $\widehat{H_2}$, $\widehat{H_3}$ are equivalent histories (they involve the same set of events) Let $\widehat{H_4}$ be the history defined as follows

$$\widehat{H_4} = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e8\ e10]\ [e7.$$

$\widehat{H_4}$ is a partial sequential history. All these histories have the same local history for process p_1: $\widehat{H}|p_1 = \widehat{H_1}|p_1 = \widehat{H_2}|p_1 = \widehat{H_3}|p_1 = \widehat{H_4}|p_1 =$ $[e1\ e3]\ [e4\ e6]\ [e8\ e10]$, and, as far p_2 is concerned, $\widehat{H_3}|p_2$ is a prefix of $\widehat{H}|p_2 = \widehat{H_1}|p_2 = \widehat{H_2}|p_2 = \widehat{H_3}|p_2 =$ $[e2\ e5]\ [e7\ e9]$.

So far, we defined the notion of a history as an abstract way to depict the interaction between a set of processes and a set of shared objects. In short, a history is a total order on the set of invocation and response events generated by the processes on the objects. We are now ready to define the notion of atomic history.

3 Atomicity

Among all possible histories of a set of processes accessing shared objects, an important question is to define which ones are correct. This section introduces the correctness condition called *atomicity* (or *linearizability*). The aim of atomicity is to transform the difficult problem of reasoning about a concurrent execution into the simpler problem of reasoning about a sequential one.

Intuitively, atomicity states that a history is correct if its invocation and response events could have been obtained, in the same order, by a single sequential process. In an atomic (also called linearizable) history, each operation has to appear as if it has been executed alone and instantaneously at some point between its invocation event and its response event. This section defines formally the atomicity concept and presents its main properties.

3.1 Legal history

As we pointed out earlier, shared objects that are usually considered in programming typically have a sequential specification defining their semantics. Not surprisingly, a definition of what is a "correct" history has to refer in one way or another to sequential specifications. The notion of *legal* history captures this idea.

Given a sequential history $\widehat{S}$, let $\widehat{S}|X$ ($\widehat{S}$ at X) denote the subsequence of $\widehat{S}$ made up of all the events involving object X. We say that a sequential history $\widehat{S}$ is *legal* if, for each object X, the sequence $\widehat{S}|X$ belongs to the sequential specification of X. In a sense, a history is legal if it could have been generated by processes sequentially accessing objects.

3.2 The case of complete histories

We first define in this section atomicity for complete histories $\widehat{H}$, i.e., histories without pending operations: each invocation event of $\widehat{H}$ has a matching response event in $\widehat{H}$. The section that follows will extend this definition to partial histories.

Definition A complete history $\widehat{H}$ is *atomic* (or *linearizable*) if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{H}$ and $\widehat{S}$ are equivalent,
2. $\widehat{S}$ is sequential and legal, and
3. $\rightarrow_H \subseteq \rightarrow_S$.

The definition above states that for a history $\widehat{H}$ to be linearizable, there must exist a permutation of $\widehat{H}$, $\widehat{S}$ (witness history), which satisfies the following requirements. First, $\widehat{S}$ has to be composed of the same set of events as $\widehat{H}$ [item 1]. Second, $\widehat{S}$ has to be sequential (interleave the process histories at the granularity of complete operations) and legal (respect the sequential specification of each object) [item 2]. Notice that, as $\widehat{S}$ is sequential, $\rightarrow_S$ is a total order. Finally, $\widehat{S}$ has also to respect the real-time occurrence order of the operations as defined by $\rightarrow_H$ [item 3]. $\widehat{S}$ represents a history that could have been obtained by executing all the operations, one after the other, while respecting the occurrence order of non-overlapping operations. Such a sequential history $\widehat{S}$ is called a *linearization* of $\widehat{H}$.

When proving that an algorithm implements an atomic object, we need to prove that all histories generated by the algorithm are linearizable, i.e., identify a linearization of its operations that respects the "real-time" occurrence order of the operations and that is consistent with the sequential specification of the object.

It is important to notice that the notion of atomicity includes inherently a form of nondeterminism. More precisely, given a history $\widehat{H}$, several linearizations of $\widehat{H}$ might exist.

Linearization: an example Let us consider the history $\widehat{H}$ described in Figure 4 where the dequeue operation invoked by p_1 returns the value b while the dequeue operation invoked by p_2 returns the value a. This means that we have $e9 = resp[Deq(a) \text{ by } p_2]$ and $e10 = resp[Deq(b) \text{ by } p_1]$. To show that this history is linearizable, we have to exhibit a witness history (linearization) satisfying the three requirements of atomicity. The reader can check that history $\widehat{H_1} = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e7\ e9]\ [e8\ e10]$ defined in Section 2.4 is such a witness. At the granularity level defined by the operations, witness history $\widehat{H_1}$ can be represented as follows

$$[Enq(a) \text{ by } p_1][Enq(b) \text{ by } p_1][Enq(c) \text{ by } p_2][Deq(a) \text{ by } p_2][Deq(b) \text{ by } p_1].$$

This formulation highlights the intuition that underlies the definition of the atomicity concept.

Linearization point The very existence of a linearization of a an atomic history $\widehat{H}$ means that each operation of $\widehat{H}$ could have been executed at an indivisible instant between its invocation and response time events (while providing the same result as $\widehat{H}$). It is thus possible to associate a *linearization point* with each operation of an atomic history. This is a point of the time-line at which the corresponding operation could have been "instantaneously" executed according to the witness sequential and legal history.

To respect the real time occurrence order, the linearization point associated with an operation has always to appear within the interval defined by the invocation event and the response event associated with that operation.

Example Figure 5 depicts the linearization point of each operation. A triangle is associated with each operation, such that the vertex at the bottom of a triangle (bold dot) represents the associated linearization point. A triangle shows how atomicity allows shrinking an operation (the history of which takes some duration) into a single point of the time-line.

In that sense, atomicity reduces the difficult problem of reasoning about a concurrent system to the simpler problem of reasoning about a sequential system where the operations issued by the processes are instantaneously executed.

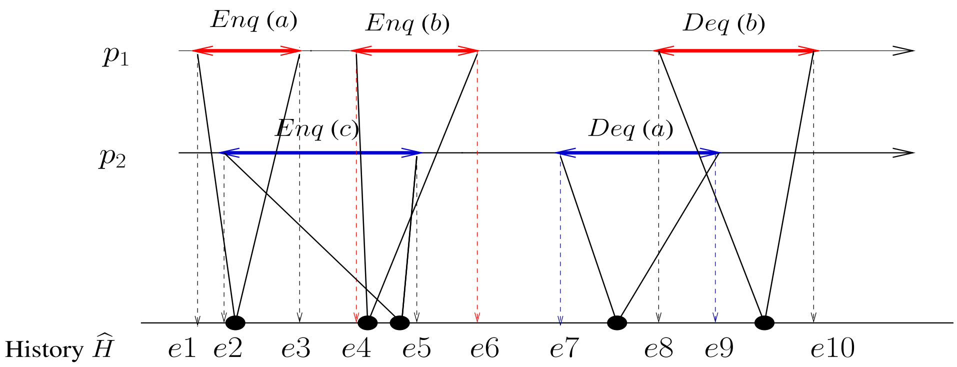

Figure 5: Linearization points

As a second example, let us again consider the history depicted in Figure 5 where the response events $e9$ and $e10$ are such that $e9 = resp[Deq(b)$ by $p_2]$ and $e10 = resp[Deq(a)$ by $p_1]$. It is easy to see that this history is linearizable: the sequential history $\widehat{H_2}$ described in Section 2.4 is a linearization. Similarly, the history where $e9 = resp[Deq(c)$ by $p_2]$ and $e10 = resp[Deq(a)$ by $p_1]$ is also linearizable. It has the following sequential witness history:

$$[Enq(c) \text{ by } p_2][Enq(a) \text{ by } p_1][Enq(b) \text{ by } p_1][Deq(c) \text{ by } p_2][Deq(a) \text{ by } p_1].$$

On the other hand, the history in which the two dequeue operations would return the same value is not linearizable: it does not have any witness history which respects the sequential specification of the queue.

3.3 The case of partial histories

We show here how to extend the definition of atomicity to partial histories. As we explained, these are histories with at least one process whose last operation is pending: the invocation event of this operation appears in the history while the corresponding response event does not. The history $\widehat{H_4}$ described in Section 2.4 is such a partial history. Extending atomicity to partial histories is important as it allows to cope with process crashes.

Definition A partial history $\widehat{H}$ is linearizable if $\widehat{H}$ can be modified in such a way that every invocation of a pending operation is either removed or completed with a response event, and the resulting (complete) history $\widehat{H'}$ is linearizable.

Basically, we reduce the problem of determining whether a partial history $\widehat{H}$ is linearizable to the problem of determining whether a complete history $\widehat{H'}$, extracted from $\widehat{H}$, is linearizable. We obtain $\widehat{H'}$ by adding response events to certain pending operations of $\widehat{H}$, as if these operations have indeed been completed, but also removing invocation events from some of the pending operations of $\widehat{H}$. We require however that all complete operations of $\widehat{H}$ be preserved in $\widehat{H'}$. It is important to notice that, given a history $\widehat{H}$, we can extract several histories $\widehat{H'}$ that satisfy the required conditions.

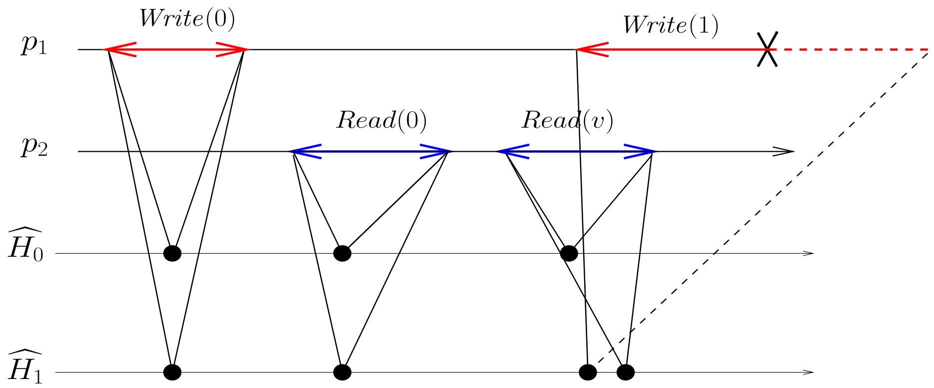

Figure 6: Two ways of completing a history

Example Consider Figure 6 where we depict two processes accessing a shared register. Process p_1 first writes the value 0. The same process later issues a write for the value 1, but p_1 crashes during this second write (this is indicated by a cross on its time-line). Process p_2 executes two consecutive read operations. The first read operation lies between the two write operations of p_1 and returns the value 0. A different value would clearly violate atomicity. The situation is less obvious with the second value and it is not entirely clear what value v has to be returned by the second read operation in order for the history to be linearizable?

As we now explain, both values 0 and 1 can be returned by that read operation while preserving atomicity. The second write operation is pending in the partial history $\widehat{H}$ modeling this execution. This history $\widehat{H}$ is made up of 7 events (the name of the object and process names are omitted as there is no ambiguity), namely:

$$inv[write(0)]\ resp[write(0)]\ inv[read(0)]\ resp[read(0)]\ inv[read(v)]\ inv[write(1)]\ resp[read(v)].$$

We explain now why both 0 and 1 can be returned by the second read:

- Let us first assume that the returned value v is 0.
  We can associate with history $\widehat{H}$ a legal sequential witness history $\widehat{H_0}$ which includes only complete operations and respects the partial order defined by $\widehat{H}$ on these operations (see Figure 6). To obtain $\widehat{H_0}$, we construct history $\widehat{H'}$ by removing from $\widehat{H}$ event $inv[write(1)]$: we obtain a complete history, i.e., without pending operations.

  History $\widehat{H}$ with $v = 0$ is consequently linearizable. The associated witness history $\widehat{H_0}$ models the situation where p_1 is considered as having crashed before invoking the second write operation: everything appears as if this write has never been issued.

- Let us now assume that the returned value v is 1.
  Similarly to the previous case, we can associate with history $\widehat{H}$ a witness legal sequential history $\widehat{H_1}$ that respects the partial order on the operations. We actually derive $\widehat{H_1}$ by first constructing $\widehat{H'}$, which we obtain by adding to $\widehat{H}$ the response event $res[write(1)]$. (In Figure 6, the part added to $\widehat{H}$ in order to obtain $\widehat{H'}$ -from which $\widehat{H_1}$ is constructed- is indicated by dotted lines).

  The history where $v = 1$ is consequently linearizable. The associated witness history $\widehat{H_1}$ represents the situation where the second write is taken into account despite the crash of the process that issued that write operation.

4 Locality

This section presents an inherent property of atomicity that makes it particularly attractive. (Another important property of atomicity, namely *non-blocking*, are discussed in the next chapters.)

4.1 Local property

Let P be any property that is on a set of objects. The property P is said to be *local* if the set of objects as a whole satisfies P whenever each object taken alone satisfies P.

Locality is an important concept that promotes modularity. Consider some local property P. To prove that an entire set of objects satisfy P, we only have to ensure that each object -independently from the others satisfies P. As a consequence, property P can be implemented on a per object basis. At one extreme, it is even possible to design an implementation where each object has its own algorithm implementing P. At another extreme, all the objects (whatever their types) might use the same algorithm to implement P (each object using its own instance of the algorithm).

4.2 Atomicity is a local property

We prove in the following that atomicity is a local property. Intuitively, the fact that atomicity is local comes from the fact that (1) it considers that each operation is on single object, and (2) it involves the real-time occurrence order on non-concurrent operations whatever the objects and the processes concerned by these operations. We will rely on these two aspects in the proof of the following theorem.

Theorem 1 *A history $\widehat{H}$ is atomic (linearizable) if and only if, for each object X involved in $\widehat{H}$, $\widehat{H}|X$ is atomic (linearizable).*

Proof The "$\Rightarrow$" direction (only if) is an immediate consequence of the definition of atomicity: if $\widehat{H}$ is linearizable then, for each object X involved in $\widehat{H}$, $\widehat{H}|X$ is linearizable. So, the rest of the proof is restricted to the "$\Leftarrow$" direction. We also restrict the rest of the proof to the case where $\widehat{H}$ is complete, i.e., $\widehat{H}$ has no pending operation. This is without loss of generality, given that the definition of atomicity for a partial history is derived from the definition of atomicity for a complete history.

Given an object X, let $\widehat{S_X}$ be a linearization of $\widehat{H}|X$. It follows from the definition of atomicity that $\widehat{S_X}$ defines a total order on the operations involving X. Let $\rightarrow_X$ denote this total order. We construct an order relation $\rightarrow$ defined on the whole set of operations of $\widehat{H}$ as follows:

1. For each object X: $\rightarrow_X \ \subseteq \ \rightarrow$,
2. $\rightarrow_H \ \subseteq \ \rightarrow$.

Basically, "$\rightarrow$" totally orders all operations on the same object object X, according to $\rightarrow_X$ (item 1), while preserving $\rightarrow_H$, i.e., the real-time occurrence order on the operations (item 2).
Claim. "$\rightarrow$ is acyclic". This means that $\rightarrow$ indeed defines a partial order on the set of all the operations of $\widehat{H}$. Assuming this claim, it is thus possible to construct a sequential history $\widehat{S}$ including all events of $\widehat{H}$ and respecting $\rightarrow$. We trivially have $\rightarrow \subseteq \rightarrow_S$ where $\rightarrow_S$ is the total order on the operations defined from $\widehat{S}$. We have the three following conditions: (1) $\widehat{H}$ and $\widehat{S}$ are equivalent (they contain the same events); (2) $\widehat{S}$ is sequential (by construction) and legal (due to item 1 above); and (3) $\rightarrow_H \subseteq \rightarrow_S$ (due to item 2 above and

$\rightarrow \subseteq \rightarrow_S$). It follows that $\widehat{H}$ is linearizable.

Proof of the claim. We show (by contradiction) that $\rightarrow$ is acyclic. Assume first that $\rightarrow$ induces a cycle involving the operations on a single object X. Indeed, as $\rightarrow_X$ is a total order, in particular transitive, there must be two operations op_i and op_j on X such that $op_i \rightarrow_X op_j$ and $op_j \rightarrow_H op_i$. But $op_i \rightarrow_X op_j \Rightarrow inv[op_i] <_H resp[op_j]$ because X is linearizable. Given that $op_j \rightarrow_H op_i \Rightarrow resp[op_j] <_H inv[op_i]$, which establishes the contradiction as $<_H$ is a total order on the whole set of events.

It follows that any cycle must involve at least two objects. To obtain a contradiction we show that, in that case, a cycle in $\rightarrow$ implies a cycle in $\rightarrow_H$ (which is acyclic). Let us examine the way the cycle could be obtained. If two consecutive edges of the cycle are due to just some $\rightarrow_X$ or just $\rightarrow_H$, then the cycle can be shortened as any of these relations is transitive. Moreover, $op_i \rightarrow_X op_j \rightarrow_Y op_k$ is not possible for $X \neq Y$, as each operation is on only one object ($op_i \rightarrow_X op_j \rightarrow_Y op_k$ would imply that op_j is on both X and Y). So let us consider any sequence of edges of the cycle such that: $op1 \rightarrow_H op2 \rightarrow_X op_3 \rightarrow_H op4$. We have:

- $op1 \rightarrow_H op2 \Rightarrow resp[op1] <_H inv[op2]$ (definition of $op1 \rightarrow_H$),
- $op2 \rightarrow_X op3 \Rightarrow inv[op2] <_H resp[op3]$ (as X is linearizable),
- $op3 \rightarrow_H op4 \Rightarrow resp[op3] <_H inv[op4]$ (definition of $op1 \rightarrow_H$).

Combining these statements, we obtain $resp[op1] <_H inv[op4]$ from which we can conclude that $op1 \rightarrow_H op4$. It follows that any cycle in $\rightarrow$ can be reduced to a cycle in $\rightarrow_H$. A contradiction as $\rightarrow_H$ is an irreflexive partial order. *End of the proof of the claim.* $\Box_{Theorem\ 1}$

Considering an execution of a set of processes that access concurrently a set of objects, atomicity allows reasoning as as the operations issued by the processes on the objects were executed one after the other. The previous theorem is fundamental. It states that when one has to reason on sequential processes that access concurrent atomic objects, one can reason on a per object basis, without loosing the atomicity property on the whole computation.

5 Alternatives to atomicity

This section discusses alternatives to atomicity, namely, *sequential consistency* and *serializability*.

5.1 Sequential consistency

Overview The definition of atomicity requires that the witness sequential history that is equivalent to a history $\widehat{H}$ respects the partial order relation on operations of $\widehat{H}$ (also called the real-time order). This is irrespective of the processes p_i and p_j and the objects Q and Q' involved in these operations. In short, sequential consistency is a weaker property that requires only for the witness history to preserve the order on the operations invoked by the same process.

Let $<_i$ denote the total order on the events generated by process p_i [1]. Moreover, let $<_{proc} = \cup_{1 \leq i \leq n} <_i$. This is the union of the n total orders associated to the processes. It is called the *process-order* relation.

To illustrate this relation, consider Figure 4, where $<_{proc}$ is the union of the local history of p_1, $<_1$, and the local history of p_2, $<_2$, namely:

$$<_1 \;=\; [e1\ e3]\ [e4\ e6]\ [e8\ e10] \qquad \text{and} \qquad <_2 \;=\; [e2\ e5]\ [e7\ e9].$$

[1]This order corresponds to the projection of $\widehat{H}$ on p_i denoted $\widehat{H}|p_i$ in Section 2.3.

It is easy to see that $\widehat{H}$ can be partitioned into two partial order relations, namely $<_{proc}$ and $<_{object}$ ($<_{object}$ is $\widehat{H}$ from which the edges due to $<_{proc}$ have been suppressed).

As we pointed out above, differently from atomicity that establishes the correctness of a history using the constraints imposed by both $<_{proc}$ and $<_{object}$, sequential consistency establishes correctness based only on the process order relation (i.e., $<_{proc}$). In other words, atomicity requires that the witness history preserves the real-time occurrence order on the events, while sequential consistency considers only local time orders (one per process) independently one from the other.

Definition The definition of the sequential consistency correctness condition reuses the notions of history, sequential history, complete history, as in Section 2. To simplify the presentation and without loss of generality, we only consider complete histories (with no pending operations). Considering $<_{proc}$ to be the process order relation on the events, we also define $\rightarrow_{proc}$ as the partial order on the operations induced from $<_{proc}$.

A history $\widehat{H}$ is *sequentially consistent* if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{H}$ and $\widehat{S}$ are equivalent,
2. $\widehat{S}$ is sequential and legal, and
3. $\rightarrow_{proc} \ \subseteq \ \rightarrow_S$ ($\widehat{S}$ has to respect process-order).

To illustrate sequential consistency, let us consider Figure 7. There are two processes p_1 and p_2 that share a queue Q. At the operation level, the local history of p_1 comprises a single operation, $Q.Enq(a)$, while the local history of p_2 comprises two operations, first $Q.Enq(b)$ and then $Q.Deq(b)$. The reader can easily verify that this history is not atomic: as all the operations are totally ordered according to real-time, the $Q.Deq()$ operation issued by p_2 should return the value a whose enqueuing was terminated before the enqueuing of a has started. However, the history is sequentially consistent: The sequential history (described at the operation level)

$$\widehat{S} = [Q.Enq(b) \text{ by } p_2][Q.Enq(a) \text{ by } p_1][Q.Deq(b) \text{ by } p_2]$$

is legal and respects the process-order relation.

Both consistency criteria, atomicity and sequential consistency, require a witness sequential history, but sequential consistency has no requirement related to the occurrence order of operations issued by different processes (and captured by the real-time order). It can be seen as based only on a logical time (the one defined by the witness history).

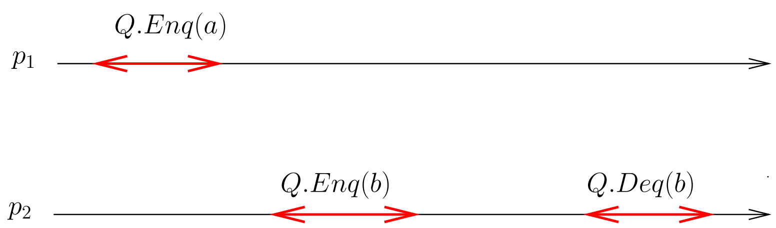

Figure 7: A sequentially consistent history

Atomicity vs sequential consistency It is easy to see from the previous definition that any linearizable history is also sequentially consistent: this is because $\rightarrow_{proc} \subseteq \rightarrow_H$. As shown by the example of Figure 7 however, the contrary is not true. It is then natural to ask whether sequential consistency would not be sufficient to judge correctness.

A drawback of sequential consistency is that it is not a local property. (This is the price that sequential consistency has to pay to allow for more correct executions than atomicity.) To illustrate this, consider the counter-example described in Figure 8. History $\widehat{H}$ involves two processes accessing two concurrent queues Q and Q'. It is easy to see that, when we consider each object in isolation, we obtain the histories $\widehat{H}|Q$ and $\widehat{H}|Q'$ that are sequentially consistent. Unfortunately, there is no way to witness a legal total order $\widehat{S}$ that involves the six operations: if p_1 dequeues b' from Q', $Q'.enq(a')$ has to be ordered after $Q'.enq(b')$ in a witness sequential history. But this means that (to respect process-order) $Q.enq(a)$ by p_1 is necessarily ordered before $Q.enq(b)$ by p_2: consequently $Q.Deq()$ by p_2 should returns a for $\widehat{S}$ to be legal. A similar reasoning can be done starting from the operation $Q.Deq(b)$ by p_2. It follows that there can be no legal witness total order: despite the fact that $\widehat{H}|Q$ and $\widehat{H}|Q'$ are sequentially consistent, the whole history $\widehat{H}$ is not.

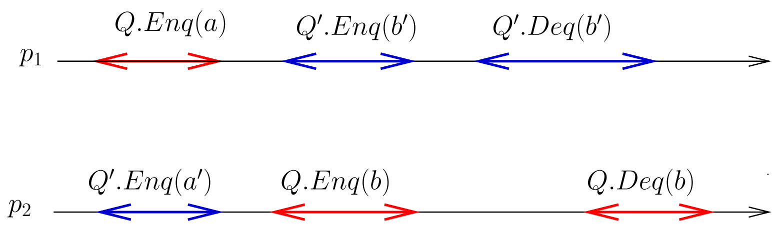

Figure 8: Sequential consistency is not a local property

5.2 Serializability

Overview Both atomicity and sequential consistency guarantee that operations appear to execute instantaneously at some point of the time line. The difference is that atomicity requires that, for each operation, this instant lies between the occurrence times of the invocation and response events associated with the operation, which is not the case for sequential consistency.

Sometimes, it is important to ensure that *groups* of operations appear to execute as if they have been executed without interference with any other group of operations. The concept of *transaction* is then the appropriate abstraction that allows grouping operations.

A transaction is a sequence of operations that might complete successfully (commit) or abort. In short, the execution of a set of concurrent transactions is correct if committed transactions appear to execute at some indivisible point in time and aborted transactions do not appear to have been executed at all. This correctness criteria is called *serializability* (sometimes it is also called atomicity). The point (again) is to reduce the difficult problem of reasoning about concurrent transactions into the easier problem of reasoning about transactions that are executed one after the other. For instance, if some invariant predicate on the set of shared objects is preserved by every individual committed transaction, then it will be preserved by a serializable execution of transactions.

Definition To define serializability, the notion of history needs to revisited. Events are now associated with objects and transactions. In short, processes are replaced by transactions. For each transaction, in addition to the invocation and response events, two new events come into the picture: *commit* and *abort* events. These are associated with transactions. At most one such event is associated with every transaction in a history. A transaction without such event is called pending; otherwise the transaction is said to be complete (committed or aborted). Adding a commit (reps., abort) event after all other events of a pending transaction is called committing (resp., aborting) the transaction. A sequential history is a sequence of committed transactions.

Let $\rightarrow_{trans}$ denote the total order of events of the committed transactions. This is the analogous to the process order relation defined above. We say that a history is complete if all its transactions are complete.

Let $\widehat{H}$ be a complete history. $\widehat{H}$ is *serializable* if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{S}$ is made of all events of committed transactions of $\widehat{H}$,
2. $\widehat{S}$ is sequential and legal, and
3. $\rightarrow_{trans} \;\subseteq\; \rightarrow_S$ ($\widehat{S}$ has to respect transaction order).

Let $\widehat{H}$ be a history that is not complete. $\widehat{H}$ is *serializable* if we can derive from $\widehat{H}$ a complete history $\widehat{H'}$ (by completing or removing pending transactions from $\widehat{H}$) such that: (1) $\widehat{H'}$ is complete, (2) $\widehat{H'}$ includes the complete transactions of $\widehat{H}$, and $\widehat{H'}$ is serializable.

Atomicity vs serializability Again, correctness is defined according to the equivalence to a witness sequential history. No real-time ordering is required. In this sense, serializability can be viewed as the extension of sequential consistency to several operations. Unlike linearizability, serializability is not a local property. Replacing in Figure 8 processes with transactions gives a counter-example that proves that.

6 Summary

We introduced in this chapter the basic elements that are needed to reason about executions of a distributed system made up of concurrent processes interacting through shared objects. More specifically, we introduced the elements that are needed to introduce the atomicity concept.

The fundamental element is that of a history: a sequence of events depicting the interaction between processes and objects. An event represents the invocation of an object or the return of a response. A history is atomic if, despite concurrency, it appears as if processes access the objects of the history in a sequential manner. In this sense, the correctness of a concurrent computation is judged with respect to a sequential behavior, itself determined by the sequential specification of the objects.

7 Bibliographic notes

The notion of atomic read/write objects (registers), as studied here, has been investigated and formalized by Lamport [8] and Misra [9]. The generalization of the atomicity consistency condition to objects of any sequential type has been developed by Herlihy and Wing under the name linearizability [6].

The notion of sequential consistency has been introduced by Lamport [7]. The relation between atomicity and sequential consistency was investigated in [2] and [11] where it was shown that, from a protocol design point of view, sequential consistency can be seen as a lazy linearizability. Examples of protocols implementing sequential consistency can be found in [1, 2, 12].

The concept of transactions is part of every textbook on database systems. Books entirely devoted to transactions are [3, 4, 5]). The theory of serializability was the main topic of the following books [4, 10].

References

[1] Afek Y., Brown G. and Merritt M., Lazy Caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182-205, 1993.

[2] Attiya H. and Welch J.L., Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91-122, 1994.

[3] Bernstein A., Haqzilacos V. and Goodman N., Concurrency Control and Recovery in Database Systems. *Addison Wesley*, 1986.

[4] Fekete A., Lynch N., Merritt M. and Weihl W., Atomic Trasanctions. *Morgan Kaufmann Publishing*, 1994.

[5] Gray J. and Reuter A., Transactions Procesing: Concepts and Techniques, *Morgan Kaufmann Publishing*, 1070 pages, 1992.

[6] Herlihy M.P. and Wing J.M, Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[7] Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.

[8] Lamport. L., On Interprocess Communication, Part 1: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77-101, 1986.

[9] Misra J., Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153, 1986.

[10] Papadimitriou C., The Theory of Database Concurrency Control. *Computer Science Press*, 1988.

[11] Raynal M., Sequential Consistency as Lazy Linearizabilty. *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, pp. 151-152, Winnipeg, 2002.

[12] Raynal M., Token-Based Sequential Consistency. *Int'l Journal of Computer Systems Science and Engineering*, 17(6):359-366, 2002.