Concurrent Computing

Rachid Guerraoui Petr Kuznetsov

January 21, 2018

Contents

1. Introduction

In 1926, Gilbert Keith Chesterton published a novel “The Return of Don Quixote” reflecting the advancing industrialization of the Western world, where mass production started replacing personally crafted goods. One of the novel’s characters, soon to be converted in a modern version of Don Quixote, says:

> ”All your machinery has become so inhuman that it has become natural. In becoming a second nature, it has become as remote and indifferent and cruel as nature. ... You have made your dead system on so large a scale that you do not yourselves know how or where it will hit. That’s the paradox! Things have grown incalculable by being calculated. You have tied men to tools so gigantic that they do not know on whom the strokes descend.”

Since mid-1920s, we made a huge progress in ’dehumanizing’ machinery, and computing systems are among the best examples. Indeed, modern large-scale distributed software systems are often claimed to be the most complicated artifacts ever existed. This complexity triggers a perspective on them as natural objects. This is, at the very least, worrying. Indeed, given that our daily life relies more and more upon computing systems, we should be able to understand and control their behavior.

In 2003, almost 80 years after the Chesterton’s book was published, Leslie Lamport, in his invited lecture “Future of Computing: Logic or Biology”, called for a reconsideration of the general perception of computing:

> ”When people who can’t think logically design large systems, those systems become incomprehensible. And we start thinking of them as biological systems. And since biological systems are too complex to understand, it seems perfectly natural that computer programs should be too complex to understand.
>
> We should not accept this. ”

In this book, we intend to support this point of view by presenting a consistent collection of basic comprehensive results in computing, and more specifically, in an important branch of it called *concurrent computing*. Concurrent computing systems are treated here as logical entities, namely algorithms, with clears goals and strategies.

1.1. A broad picture: the concurrency revolution

The field of *concurrent computing* has gained a huge importance after major chip manufacturers have switched their focus from increasing the speed of individual processors to increasing the number of processors on a chip. The good old days where nothing needed to be done to boost the performance of programs, besides changing the underlying processors, are over. A single-threaded application can exploit at most 1/100 of the potential throughput of a 100-core chip. To exploit multicore architectures, programs have to be executed in a concurrent manner. The algorithms have to be designed with a large number of *threads* (also called *processes*) and we need to make sure that their concurrent accesses to shared data do not create inconsistencies.

The computer industry is thus calling for a software revolution: the *concurrency revolution*. This might look surprising at first glance for the very idea of concurrency is almost as old as computer science.

In fact, the revolution is more than about concurrency alone: it is about *concurrency for everyone*. Concurrency is going out of the small box of specialized programmers and is conquering the masses now. Somehow, the very term ”concurrency” itself captures this democratization: we used to talk about ”parallelism”. Specific kinds of programs designed by specialized experts to clearly involve independent tasks were deployed on parallel architectures. The term ”concurrency” better reflects a wider range of programs where facts that the tasks executing in parallel compete for shared data is the norm rather than the exception. But designing and implementing such programs in a correct and efficient manner is not trivial.

A major challenge underlying the concurrency revolution is to come up with a *library of abstractions* that programmers can use for general purpose concurrent programming. Ideally, such library should both be usable by programmers with little expertise in concurrent programming as well as by advanced programmers who master how to leverage multicore architectures. The ability of these abstractions to be composed is of key importance, for an application is typically the result of assembling independently devised pieces of code.

The aim of this book is to study how to define and build such abstractions. We will focus on those that are considered (a) the most difficult to get right and (b) having the highest impact on the overall performance of a program: *synchronization abstractions*, also called *shared objects* or sometimes *concurrent data structures*.

1.2. The topic: shared objects

In concurrent computing, a problem is solved through several threads (processes) that execute a set of tasks. In general, and except in so called ”embarrassingly parallel” programs, i.e., programs that solve problems that can easily and regularly be decomposed into independent parts, the tasks usually need to synchronize their activities by accessing shared constructs, i.e., these tasks depend on each other. These typically serialize the threads and reduce parallelism. According to Amdahl’s law [4], the cost of accessing these constructs significantly impacts the overall performance of concurrent computations. Devising, implementing and making good usage of such synchronization elements usually lead to intricate schemes that are very fragile and sometimes error prone.

Every multicore architecture provides synchronization constructs in hardware. Usually, these constructs are “low-level” and making good usage of them is not trivial. Also, the synchronization constructs that are provided in hardware differ from architecture to architecture, making concurrent programs hard to port. Even if these constructs look the same, their exact semantics on different machines may also be different, and some subtle details can have important consequences on the performance or the correctness of the concurrent program. Clearly, coming up with a high-level library of synchronization abstractions that could be used across multicore architectures is crucial to the success of the multicore revolution. Such a library could only be implemented in software for it is simply not realistic to require multicore manufacturers to agree on the same high-level library to offer to their programmers.

We assume a small set of low-level synchronization primitives provided in hardware, and we use these to implement higher level synchronization abstractions. As pointed out, these abstractions are supposed to be used by programmers of various skills to build application pieces that could themselves be used within a higher-level application framework.

The quest for synchronization abstractions, i.e., the topic of this book, can be viewed as a continuation of one of the most important quests in computing: programming *abstractions*. Indeed, the History of computing is largely about devising abstractions that encapsulate the specifities of underlying hardware and help programmers focus on higher level aspects of software applications. A *file*, a *stack*, a *record*, a *list*, a *queue* and a *set*, are well-known examples of abstractions that have proved to be valuable in traditional sequential and centralized computing. Their definitions and effective implementations have enabled programming to become a high-level activity and made it possible to reason about algorithms without specific mention of hardware primitives.

In modern computing, an abstraction is usually captured by an *object* representing a server program that offers a set of operations to its users. These operations and their specification define the behavior of the object, also called the *type* of the object.

The way an abstraction (object) is implemented is usually hidden to its users who can only rely on its operations and their specification to design and produce upper layer software, i.e., software using that object. The only visible part of an object is the set of values in can return when its operations are invoked. Such a modular approach is key to implementing provably correct software that can be reused by programmers in different applications.

The abstractions we study in this book are *shared* objects, i.e., objects that can be accessed by concurrent processes, typically running on independent processors. That is, the operations exported by the shared object can be accessed by concurrent processes. Each individual process accesses however the shared object in a sequential manner. Roughly speaking, sequentiality means here that, after it has invoked an operation on an object, a process waits to receive a reply indicating that the operation has terminated, and only then is allowed to invoke another operation on the same or a different object. The fact that a process p is executing an operation on a shared object X does not however preclude other processes q from invoking an operation on the same object X.

The objects considered have a precise *sequential specification.* called also its *sequential type*, which specifies how the object behaves when accessed sequentially by the processes. That is, if executed in a sequential context (without concurrency), their behavior is known. This behavior might be deterministic in the sense that the final state and response is uniquely defined given every operation, input parameters and initial state. But this behavior could also be non-deterministic, in the sense that given an initial state of the object, and operation and an input parameter, there can be several possibilities for a new state and response.

To summarize, this book studies how to implement, in the algorithmic sense, objects that are shared by concurrent processes. Strictly speaking, the objective is to implement object types but when there is no ambiguity, we simply say objects. In a sense, a process represents a sequential Turing machine, and the system we consider represents a set of sequential Turing machines. These Turing machines communicate and synchronize their activities through low-level shared objects. The activities they seek to achieve consist themselves in implementing higher-level shared objects. Such implementations need to be *correct* in the sense that they typically need to satisfy two properties: *linearizability* and *wait-freedom*. We now overview these two properties before detailing them later.

1.3. Correctness (Part I): Linearizability

In short, *linearizability* says that, despite concurrency among (processes invoking) operations of an object, these should *appear* as if they were executed *sequentially*. Two concepts are important here. The first is the notion of *appearance*, which, as we already pointed out, is related to the values returned by an operation: these values are the only way through which the behavior of an object is visible to the users of that object, i.e., the applications using that object. The second is the notion of *sequentiality* which we also discussed earlier. Namely, The operations issued by the processes on the shared objects should appear, according to the values they return, as if they were executing one after the other. Each operation invocation *op* on an object X should appear to take effect at some indivisible instant, called the *linearization* point of that invocation, between the invocation and the reply times of *op*.

In short, linearizabiliy delimits the scope of an object operation, namely what it could respond in a concurrent context, given the sequential specification of that object.

This property, also sometimes called *atomicity*, transforms the difficult problem of reasoning about a concurrent system into the simpler problem of reasoning about a sequential one where the processes access each object one after the other.

Linearizability constraints the implementation of the object but simplifies its usage on the other hand. To program with linearizable objects, also called atomic objects, the developer simply needs the *sequential specification* of each object, i.e., its sequential type.

Most interesting synchronization problems are best described as linearizable shared objects. Examples of popular synchronization problems are the *reader-writer* and the *producer-consumer* ones. In the reader-writer problem, the processes need to read or write a shared data structure such that the value read by a process at a given point in time t is the last value written before t. Solving this problem boils down to implementing a linearizable object exporting *read*() and *write*() operations. Such an object type is usually called a linearizable, an atomic read-write variable, or a *register*. It abstracts the very notions of shared file and disk storage.

In the producer-consumer problem, the processes are usually split into two camps: the producers which create items and the consumers which use the items. It is typical to require that the first item produced is the first to be consumed. Solving the producer-consumer problem boils down to implementing a linearizable object type, called a FIFO *queue* (or simply a queue) that exports two operations: *enqueue*() (invoked by a producer) and *dequeue*() (invoked by a consumer).

Other exemples include for instance *counting*, where the problem consists in implementing a shared counter, called FAI *Fetch-and-Increment*. Processes invoque this object to increment the value of the counter and get the current value.

1.4. Correctness (Part II): Wait-freedom

Wait-freedom basically says that processes should not prevent each other from performing operations and obtaining corresponding responses. More specifically, no process p should ever prevent any other process q from making progress, i.e., obtaining responses to q's operations, provided q remains alive and kicking. A process q should be able to terminate each of its operations on a shared object X despite speed variations or the failure of any other process p. Process p could be very fast and might be permanently accessing shared object X, or could have been swapped out by the operating system while accessing X. None of these situations should prevent q from completing its operation.

Wait-freedom conveys the *robustness* of an implementation. It is a *liveness*, also called a *progress*, property.

Wait-freedom transforms the difficult problem of reasoning about a failure-prone system where processes can be arbitrarily delayed or speeded up, into the simpler problem of reasoning about a system where every process progresses at its own pace and runs to completion.

In other words, wait-freedom says that the process invoking the operation on the object should obtain a response for the operation, in a finite number of its own *steps*, independently of concurrent steps from other processes. The notion of step, as we will discuss later, means here a local instruction of the process, say updating a local variable, or an operation invocation on a base object (low-level object) used in the implementation.

Ensuring linearizability alone or wait-freedom alone is simple. A trivial wait-free implementation could return arbitrary responses to each operation, say some value corresponding to some initial state of the object. This would satisfy wait-freedom as no process would prevent other processes from progressing. However, the responses would no satisfy linearizability.

Also, one could ensure linearizability using a basic *mutual exclusion* mechanism so that every operation on the implemented object is performed in an indivisible critical section. Some traditional synchronization schemes rely indeed on *mutual exclusion* (usually based on some *locking* primitives): critical shared objects (or critical sections of code within shared objects) are accessed by processes one at a time. No process can enter a critical section if any other process is in that critical section. We also say that a process has acquired a *lock* on that object (resp., critical section). Linearizability is then automatically ensured if all related variables are protected by the same critical section. This however significantly limits the parallelism and thus the performance of the program, unless the program is devised with minimal interference among processes. Mutual exclusion hampers progress since a process delayed in a critical section prevents all other processes from entering that critical section. In other words, it violates wait-freedom. Delays could be significant and especially when caused by crashes, preemptions and memory paging. For instance, a process paged-out might be delayed for millions of instructions, and this would mean delaying many other processes if these want to enter the critical section held by the delayed process. With modern architectures, we might be talking about one process delaying hundreds of processors, making them completely idle and useless.

Sometimes, as we shall see, wait-freedom is considered very expensive. We will study other, weaker *lock-free* implementations, which also provide an alternative to mutual exclusion-based implementations.

1.5. Reducibility of Algorithms

As explained, this book studies how to wait-free implement high-level atomic objects out of more primitive base objects. The notions of *high* and *primitive* being of course relative as we will see. It is also important to notice that the term *implement* is to be considered in an abstract manner; we will describe the concurrent algorithms in pseudo-code. There will not be any C or Java code in this book. A concrete execution of these algorithms would need to go through a translation into some programming language.

An object to be implemented is typically called *high-level*, in comparison with the objects used in the implementation, considered at a *lower-level*. It is common to talk about *emulations* of the high-level object using the low-level ones. Unless explicitly stated otherwise, we will by default mean *wait-free implementation* when we write *implementation*, and *atomic (linearizable) object* when we write *object*.

It is often assumed that the underlying system provides some form of *registers* as base objects. These provide the abstraction of read-write storage elements. Message-passing systems can also, under certain conditions, emulate such *registers*. Sometimes the base *registers* that are supported are atomic but sometimes not. As we will see in this book, there are algorithms that implement atomic *registers* out of non-atomic base *registers* that might be provided in hardware.

Some multiprocessor machines also provide objects that are more powerful than *registers* like *test&set* objects or *compare&swap* objects. These are more powerful in the sense that the writer process does not systematically overwrite the state of the object, but specifies the conditions under which this can be done. Roughly speaking, such conditional update enables more powerful synchronization schemes than with a simple *register* object. We will capture the notion of "more powerful" more precisely later in the book.

In its abstract form, the question we address in this book, namely of implementing high-level objects using lower level objects, can be stated as a general *reducibility* question. Given two object types $X1$ and $X2$, can we implement $X2$ using any number of instances of $X1$ (we simply say "using $X1$")? In other words, is there an algorithm that implements $X2$ using $X1$? In the case of concurrent computing, "implementing" typically assumes providing linearizability and wait-freedom. These notions encapsulate the smooth handling of concurrency and failures.

When the answer to the reducibility question is negative, and it will be for some values of $X1$ and $X2$, it is also interesting to ask what is needed (under some minimality metric) to add to the low-level objects ($X1$) in order to implement the desired high-level object ($X2$). For instance, if the base objects provided by a given multiprocessor machine are not enough to implement a particular object in software, knowing that extending the base objects with another specific object (or many of such objects) is sufficient, might give some useful information to the designers of the new version of the multiprocessor machine in question. We will see examples of these situations.

1.6. Organization

The book is organized in an incremental way, starting from very basic objects, then going step by step to implementing more and more sophisticated and powerful objects.

1. We start by defining precisely the very notions of linearizability and wait-freedom. In order to do so, we define the concepts of histories, modeling executions, as well as what it means for a history to be linearizable. Then we distinguish the notions of low-level and high-level histories and introduce progress properties. In particular, we define the concept of wait-freedom.

2. We then study how to implement linearizable read-write *registers* out of non-linearizable base *registers*, i.e., *registers* that provide weaker guarantees than linearizability. Furthermore, we show how to implement *registers* that can contain values from an arbitrary large range, and be read and written by any process in the system, starting from single-bit (containing only 0 or 1) base *registers*, where each base *register* can be accessed by only one writer process and only one reader process. Many of these algorithms are simple but contain fundamental ideas that we encouter when implementing more sophisticated objects.

3. We then discuss how to use *registers* to implement seemingly more sophisticated objects like *snapshot* objects. In short, a *snapshot* captures the entire state of a system of processes. When such an object is atomic, it conveys an instantaneous picture of the state of the system. Implementing it in a wait-free manner is not trivial and we will present algorithmic techniques, that are interesting by themselves, to achieve that. We will also discuss various forms of emphsnapshots and what algorithmic schemes are needed to implement them.

4. We will then turn to explain the inherent limitation of *registers* in implementing more powerful objects like *queues* or *stacks*. This limitation is highlighted through the seminal *consensus impossibility* result. In short, not subset of at least two processes can, using only *registers*, wait-free implement the *consensus* object, and as a corollary, can implement objects like *queues* or *stacks*. This result is a central one in concurrent and distributed computing and we will present it in a detailed manner.

5. We then discuss the importance of *consensus* as an object type, by proving its *universality*. In particular, we describe a simple algorithm that uses *registers* and *consensus* objects to implement any other object. We then turn to the question of how to implement a *consensus* object from other objects. We describe an algorithm to implement a *consensus* object in a system of two processes, using registers and either a *test&set* or a *queue* objects, as well as an algorithm that implements a *consensus* object using a *compare&swap* object in a system with an arbitrary number of processes. The difference between these implementations is highlighted to introduce the notion of *consensus number*.

6. We then study a complementary way of achieving *universality*: using *registers* and specific oracles that reveal certain information about the operational status of the processes. Such oracles can be

viewed as *failure detectors* providing information about which process are operational and which processes are not. We discuss how even an oracle that is unreliable most of time can help devise a *consensus* algorithm and hence achieve *universality*. We also discuss the implementation of such an oracle assuming that the computing environment satisfies additional assumptions about the scheduling of the processes. This may be viewed as a slight weakening of the wait-freedom requirement which requires progress no matter how processes interleave their steps.

1.7. Bibliographical notes

The fundamental notion of abstract object type has been developed in various textbooks on the theory or practice of programming. Early works on the genesis of abstract data types were described in [19, 53, 60, 59]. In the context of concurrent computing, one of the earliest work was reported in [15, 58]. More information on the history concurrent programming can be found in [13].

The concept of *register* (as considered in this book) and its formalization are due to Lamport [49]. A more hardware-oriented presentation was given in [57]. The notion of atomicity has been generalized to any object type by Herlihy and Wing [40] under the name linearizability. The concept of *snapshot* object has been introduced in [1]. A theory of wait-free atomic objects was developed in [43].

The classical (non-robust) way to ensure linearizability, namely through mutual exclusion, has been introduced by Dijkstra [22]. The problem constituted a basic chapter in nearly all textbooks devoted to operating systems. There was also an entire monograph solely devoted to the mutual exclusion problem [63]. Various synchronization algorithms are also detailed in [66].

The property of wait-free computation originated in the work of Lamport [45], and was then explored further by Peterson [62]. It has then been generalized and formalized by Herlihy [34].

The consensus problem was introduced in [61]. Its impossibility in asynchronous message-passing systems prone to process crash failures has been proved by Fischer, Lynch and Paterson in [27]. Its impossibility in shared memory systems was proved in [55]. The universality of the consensus problem and the notion of consensus number were investigated in [34].

The concept of failure detector oracle has been introduced by Chandra and Toueg [16]. A survey to failure detectors can be found in [28].

Part I.

Correctness

2. Linearizability

2.1. Introduction

Linearizabiliy is a metric of the correctness of a shared object implementation. It addresses the question of what values can be returned by an object that is shared by concurrent processes. If an object returns a response, linearizability says whether this response is *correct* or not.

The notion of *correctness*, as captured by linearizability, is defined with respect to how the object is expected to react when accessed sequentially: this is called the *sequential specification* of the object. In this sense, the notion of correctness of an object, as captured by linearizability, is *relative* to how the object is supposed to behave in a sequential world.

It is important to notice that linearizability does not say when an object is expected to return a response. As we will see later, the complementary to linearizability is the *wait-freedom* property, another correctness metric that captures the fact that an object operation *should* eventually return a response (if certain conditions are met).

To illustrate the notion of linearizability, and the actual relation to a sequential specification, consider a FIFO (first-in-first-out) queue. This is an object of the type queue that contains an ordered set of elements and exhibits the following two operations to manipulate this set.

- *Enq*(a): Insert element a at the end of the queue;
- *Deq*$()$: Return the first element inserted in the queue that was not already removed; Then remove this element from the queue; if the queue is empty, return the default element $\perp$.

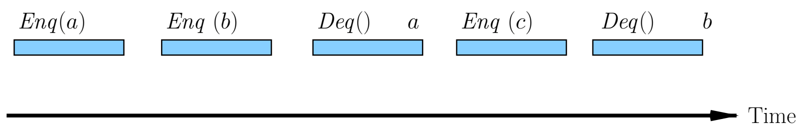


Figure 2.1.: Sequential execution of a queue

Figure 2.1 conveys a sequential execution of a system made up of a single process accessing the queue (here the time line goes from left to right). There is only a single object and a single process so we omit their identifiers here. The process first enqueues element a, then element b, and finally element c. According to the expected semantics of a queue (first-in-first-out), and as depicted by the figure, the first dequeue invocation returns element a whereas the second returns element b.

Figure 2.2 depicts a concurrent execution of a system made up of two processes sharing the same queue: p_1 and p_2. Process p_2, acting as a producer, enqueues elements $a, b, c, d,$ and then e. On the other hand, process p_1, acting as a consumer, seeks to de dequeue two elements. On Figure 2.2, the execution of $Enq(a)$, $Enq(b)$ and $Enq(c)$ by p_2 overlaps with the first $Deq()$ of p_1 whereas the execution of $Enq(c)$, $Enq(d)$ and $Enq(e)$ by p_2 overlaps with the second $Deq()$ of p_1. The questions raised in the figure are what elements can be dequeued by $p1$. The role of linearizability is precisely to address such questions.

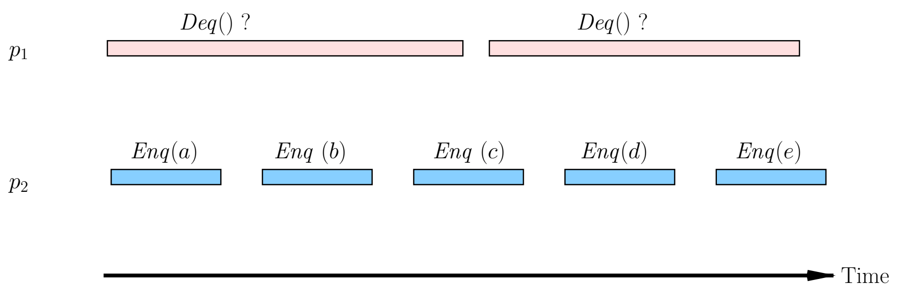


Figure 2.2.: Concurrent execution of a queue

Linearizability does so by relying on how the queue is supposed to behave if accessed sequentially. In other words, what should happen in Figure 2.2 depends on what happens in Figure 2.1. Intuitively, linearizability says that, when accessed concurrently, an object should return the same values that it could have returned in some sequential execution. Before defining linearizability however, and the very concept of "value that could have been returned in some sequential execution", we first define more precisely some important underlying elements, namely processes and objects, and then the very notion of a sequential specification.

2.2. Players

Two categories of players are important in this context, processes and objects, and these are related by the very notion of a history.

2.2.1. Processes

We consider a system consisting of a finite set of n *processes*, denoted $p_1, \dots, p_n$. Besides accessing local variables, processes may execute operations on *shared objects* (we will sometimes simply say *objects*. Through these objects, the processes *synchronize* their computations. In the context of this chapter, which aims at defining linearizability of the objects, we will omit the local variables accessed by the processes.

An execution by a process of an operation on a 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. It is common to 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 (*response event*). We say that these events are generated by process p_i and associated with 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 $p_i]$. Sometimes, when there is no ambiguity, we talk about *operations* where we should be talking about *operation executions*. We also say sometimes that the object returns a response to the process. This is by language abuse because it is actually the process executing the operation on the object that actually computes the response.

Every interaction between a process and an object 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 such events is called a *history* and this is precisely how we model executions of processes on shared objects. Basically, a history depicts the sequence of observable events of the execution of a concurrent system. We will detail the very notion of history later in this chapter.

While we assume that the system of processes is *concurrent*, we assume that each process is individually *sequential*: a process executes (at most) one operation on 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 returned, the process does not invoke any other operation. As pointed out, the fact that processes are (individually) sequential does not preclude them from concurrently invoking operations on the same shared object. Sometimes however, 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.

2.2.2. Objects

An object has a unique *identity*. The object does also have a *type*. Multiple objects can be of the same type: we talk about *instances* of the type. Strictly speaking, an object can have several types, in the case of multiple inheritance and subytping. However, for simplicity of presentation but without loss of generality, we restrict our study in this manuscript to a single type per object.

We define a type by (1) the set of possible values for (the states of) objects of that type, including the *initial* state; (2) a finite set of operations through which the (state of the) objects of that type can be manipulated; and (3) a *sequential specification* describing, for each operation of the type, the effect this operation produces when it executes alone on the object, i.e., in the absence of concurrency. The effect is measured in terms of the response that the object returns and the new state that the object gets to after the operation executes. The operations of the type are somehow said to be exported by the type.

We assume here that every operation of an object type can be applied on each of its states in the absence of concurrency. (We talk about *complete* operations or *complete* objects). This sometimes requires specific care when defining the objects. For instance, if a dequeue operation is invoked on a queue which is in an empty state, a specific response *nil* is returned.

We say that an object operation is *deterministic* if, given any state of the object and input parameters, the response and the resulting state of the object are *uniquely* defined (again in the absence of concurrency). An object type is deterministic if it has only deterministic operations. We assume here *finite* non-determinism, i.e., for each state and operation, the set of possible outcomes (response and resulting state) is finite. Otherwise the object is said to be *non-deterministic*: several outputs and resulting states are possible. The pair composed of (a) the output returned and (b) the resulting state, is chosen randomly from the set of such possible pairs (or from an infinite set).

A *sequential specification* of an object is generally modeled as a set of sequences of invocations (to that object) immediately followed by matching responses that, starting from an initial state of the object, are allowed by the object (type) when it is accessed sequentially. Indeed the resulting state obtained after each operation execution is not directly conveyed, but it is indirectly reflected through the responses returned in the subsequence operations of the sequence.

To illustrate the notion of a sequential specification, we consider two classical examples below:

Example 1: a FIFO queue Our first example is still the unbounded (FIFO) queue described earlier. The producer enqueues items in a queue that the consumers dequeues. The sequential specification is the set of euqneue and dequeue operations such that 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 *nil* is returned. As pointed out earlier this specification never prevents an enqueue or a dequeue operation to be executed. One could consider a variant of the specification where the dequeue could not be executed if the queue is empty - it would have to wait for an enqueue - we preclude such specifications.

Designing algorithms that implement this object correctly in a concurrent context captures the classical *producer/consumer* synchronization problem.

Example 2: a read/write object (register) Our second example (called register) is a simple read/write abstraction that models objects such as a shared memory word, a shared file or a shared disk. Designing algorithms that implement this object correctly in a concurrent context captures the classical *reader/writer* synchronization problem.

This type exports two operations:

- The operation $read()$ has no input parameter. It returns the value stored in the object.
- The operation $write(v)$ has an input parameter, v, representing the new value of the object. This operation returns 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 input parameter of the last preceding write operation (i.e., the last value written). We will study implementations of this object in the next chapters.

2.2.3. Histories

Processes interact with shared objects via invocation and response events. Such events are totally ordered. (Simultaneous events are arbitrarly ordered).

The interaction between processes and objects is thus modeled as a totally ordered set of events H, and is called a *history* (sometimes also called a *trace*). The total order relation on H, denoted $<_H$, abstracts out the real-time order in which the events actually occur.

Recall that an event includes (a) the name of an object, (b) the name of a process, (c) the name of an operation as well as the corresponding input or output parameters.

A *local* history of p_i, denoted $H|p_i$, is a projection of H on process p_i: the subsequence H consisting of the events generated by p_i.

Two histories H and H' are said to be *equivalent* if they have the same local histories, i.e., for each process p_i, $H|p_i = H'|p_i$.

As we consider sequential processes, we focus on histories H such that, for each process p_i, $H|p_i$ (the local history generated by p_i) is sequential: the history starts with an invocation, followed by a response, (the matching response associated with the same object) followed by another invocation, etc. We say in this case that the global history H is *well-formed.*

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. If the history contains only the invocation, we say that the operation is *pending* in that history. A history without pending operations is said to be *complete*. A history with pending operations is said to be *incomplete*. Incomplete histories are important to study as they typically model the situation where a process invokes an operation and stops, e.g., crashes, before obtaining a response. Note that, being sequential, a process can have at most one pending operation in a given history.

A history H induces an irreflexive partial order on its operations. Let $op = X.op1()$ by p_i and $op' = Y.op2()$ by p_j be two any 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 precisely:

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

Two operations *op* and *op'* are said to *overlap* (we also say they are *concurrent*) in a history H if neither $resp[op] <_H inv[op']$, nor $resp[op'] <_H inv[op]$ (neither precedes the other one). Notice that two overlapping operations are such that $\neg(op \rightarrow_H op')$ and $\neg(op' \rightarrow_H op)$. As sequential histories have no overlapping operations, $\rightarrow_H$ is a total order if H is a sequential history.

Figure 2.3 highlights the events involved in the history depicting the execution of Figure 2.2 above. The history contains events $e_1 \dots e_{14}$. As all events in H involve the same object, the identity of this object is omitted. The history has no pending operations, and is consequently complete.

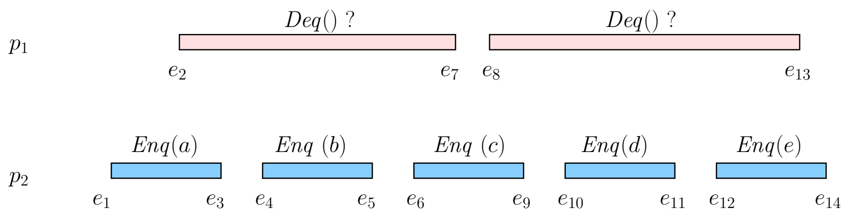


Figure 2.3.: Example of a queue history

If we restrict the history to the sequence of events $e_1 \dots e_{12}$, we will obtain an incomplete one: the last dequeue operation of p_1 as well as the last enqueue of p_2 are now pending operations in the resulting history.

2.2.4. Sequential histories

Definition 1 *A sequential history is one of which the first event is an invocation, and then (1) each invocation event, except possibly the last, is immediately followed by the matching response event, (2) each response event, except possibly the last, is immediately followed by an invocation event.*

The precision "except possibly the last" is crucial a history can be incomplete as we discussed earlier. A history that is not sequential is said to be *concurrent*.

Given that a sequential history 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.

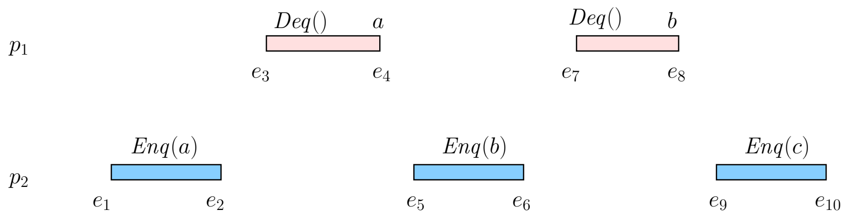


Figure 2.4.: Example of a sequential history

Figure 2.4 depicts a complete sequential history. This history has no overlapping operations. The operations are totally ordered.

2.2.5. Legal histories

As we pointed out, the definition of a linearizable history refers to the sequential specifications of the objects involved in the history. The notion of a *legal* history captures this relation.

Given a sequential history H and an object X, $H|X$ denotes the subsequence of H made up of all the events involving only object X. We say that H is *legal* if, for each object X involved in H, $H|X$ belongs to the sequential specification of X. Figure 2.4 for instance depicts a legal history. It belongs to the sequential specification of the queue. The first dequeue by p_1 returns a a whereas the second returns a b.

2.3. Linearizability

Essentially, linearizability says that a history is correct if the response returned to all operation invocations could have been obtained by a sequential execution, i.e., according to the sequential specifications of the objects. More specifically, we say that a history is linearizable if each operation appears as if it has been executed instantaneously at some indivisible point between its invocation event and its response event. This point is called the *linearization point* of the operation. We define below more precisely linearizability as well as some of its main characteristics.

2.3.1. Complete histories

For pedagogical reasons, it is easier to first define linearizability for complete histories H, i.e., histories without pending operations, and then extend this definition to incomplete histories.

Definition 2 *A* complete *history H is linearizable if there is a history L such that:*

1. *H and L are equivalent,*
2. *L is sequential,*
3. *L is legal, and*
4. *$\rightarrow_H \subseteq \rightarrow_L$.*

Hence, a history H is linearizable if there exist a permutation of H, L, which satisfies the following requirements. First, L has to be indistinguishable from H to any process: this is the meaning of equivalence. Second, L should not have any overlapping operations: it has to be sequential. Third, the restriction of L to every object involved in it should belong to the sequential specification of that object: it has to be legal. Finally, L has to respect the real-time occurrence order of the operations in H.

In short, L represents a history that could have been obtained by executing all the operations of H, one after the other, while respecting the occurrence order of non-overlapping operations in H. Such a sequential history L is called a *linearization* of H or a *sequential witness* of H.

An algorithm implementing some shared object is said to be linearizable if all histories generated by the processes accessing the object are linearizable. Proving linearizability boils down to exhibiting, for every such history, a linearization of the history that respects the "real-time" occurrence order of the operations in the history, and that belongs to the sequential specification of the object. This consists in determining for every operation of the history, its linearization point in the corresponding sequential witness 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. It is also important to notice that a history H, may have multiple possible linearizations.

Example with a queue. Consider history H depicted on Figure 2.3. Whether H is linearizable or not depends on the values returned by the dequeue invocations of p_1, i.e., in events e_7 and e_{13}. For example, assuming that the queue is initially empty, two possible values are possible for e_7: a and *nil*.

1. In the first case, depicted on Figure 2.5, the linearization of the first dequeue of p_1 would be before the first enqueue of p_2. We depict a possible linearization on Figure 2.6.

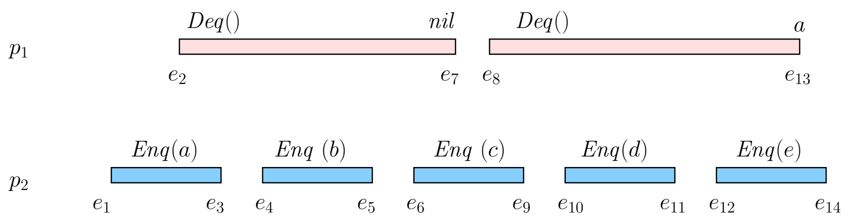


Figure 2.5.: The first example of a linearizable history with a queue

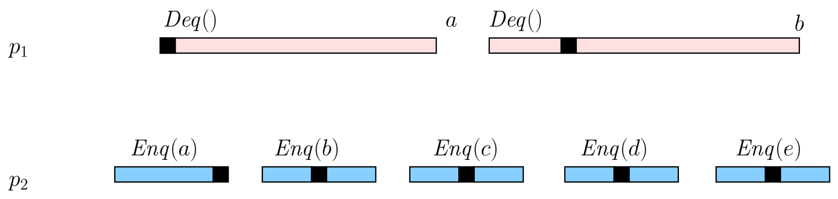


Figure 2.6.: The first example of a linearization

2. In the second case, depicted on Figure 2.7, the linearization of the first dequeue of p_1 would be after the first enqueue of p_2. We depict a possible linearization on Figure 2.8.

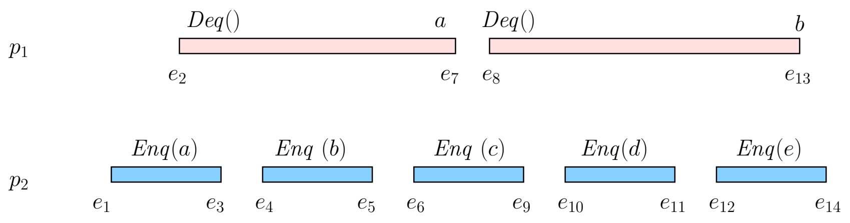


Figure 2.7.: The second example of a linearizable history with a queue

It is important to notice that, in order to ensure linearizability, the only possible values for e_7 are a and *nil*. If any other value was returned, the history of Figure 2.7. would not have been linearizable. For instance, if the value was b, i.e., if the first dequeue of p_1 returned b, then we could not have found any possible linearization of the history. Indeed, the dequeue should be linearizable after the enqueue of b, which is after the enqueue of a. To be legal, the linearization should have a dequeue of a before the dequeue of b—a contradiction.

Example with a register. Figure 2.9 highlights a history of two processes accessing a shared register. The history contains events $e_1 \dots e_{12}$. The history has no pending operations, and is consequently complete.

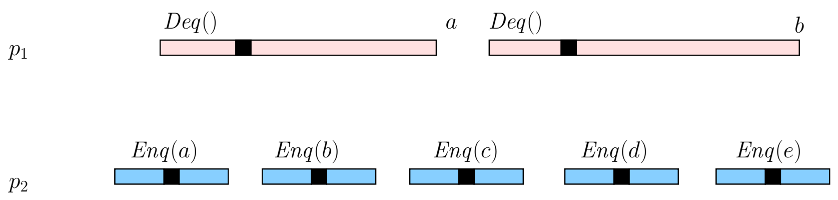


Figure 2.8.: The second example of linearization

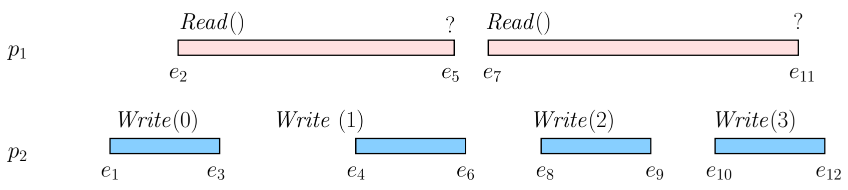


Figure 2.9.: Example of a register history

Assuming that the register initially stores value 0, two possible returned values are possible for e_5 in order for the history to be linearizable: 0 and 1. In the first case, the linearization of the first read of p_1 would be right after the first write of p_2. In the second case, the linearization of the first read of p_1 would be right after the second write of p_2.

For the second read of p_1, the history is linearizable, regardless of whether the second read of p_1 returns values 1, 2 or 3 in event e_7. If this second read had returned a 0, the history would not be linearizable.

2.3.2. Incomplete histories

So far we considered only complete histories. 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. Extending linearizability to incomplete histories is important as it allows to state what responses are correct when processes crash. We cannot decide when processes crash and then cannot expect from a process to first terminate a pending operation before crashing.

Definition 3 *A history H (whether it is complete or not) is* linearizable *if H can be* completed *in such a way that every invocation of a pending operation is either removed or completed with a response event, so that the resulting (complete) history H' is linearizable.*

Basically, this definition transforms the problem of determining whether an incomplete history H is linearizable to the problem of determining whether a complete history H', obtained by completing H, is linearizable. H' is obtained by adding response events to certain pending operations of H, as if these operations have indeed been completed, or by removing invocation events from some of the pending operations of H. (All complete operations of H are preserved in H'.) It is important to notice that the term ”complete” is here a language abuse as we might ”complete” a history by removing some of its pending invocations. It is also important to notice that, given an incomplete history H, we can complete it in several ways and derive several histories H' that satisfy the required conditions.

Example with a queue. Figure 2.10 depicts an incomplete history H. We can complete H by adding to it the response b to the second dequeue of p_1 and a response to the last enqueue of p_2: we would obtain history H' of Figure 2.5 which is linearizable. We could also have "completed" H by removing any of the pending operations, or both of them. In all cases, we would have obtained a complete history that is linearizable.

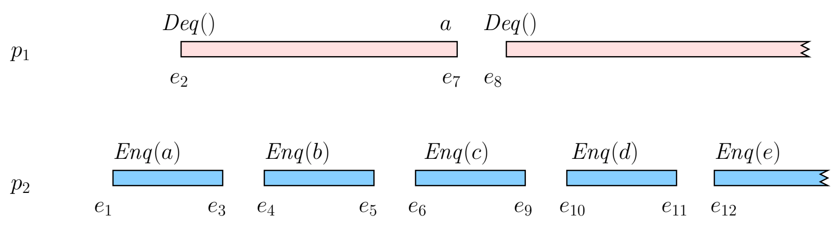


Figure 2.10.: A linearizable incomplete history

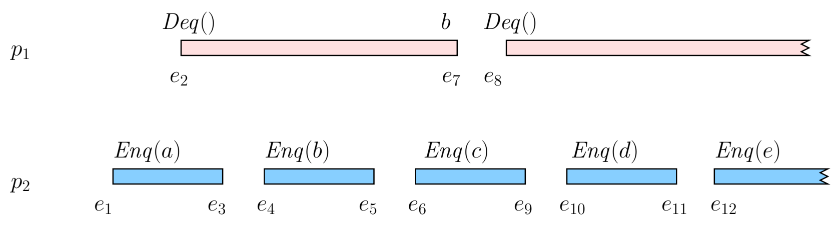


Figure 2.11.: A non-linearizable incomplete history

Figure 2.11 also depicts an incomplete history. However, no matter how we try to complete it, either by adding responses or removing invocations, there is no way to determine a linearization of the completed history.

Example with a register. Figure 2.12 depicts an incomplete history of a register. The only way to complete the history in order to make it linearizable is to complete the second write of p_2. This would enable the read of p_1 to be linearized right after it.

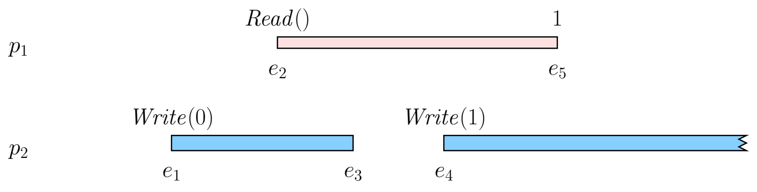


Figure 2.12.: A linearizable incomplete history of a register

2.3.3. Completing a linearizable history

An interesting characteristic of linearizability is its *nonblocking* flavour: every pending operation in a history H can be completed without having to wait for any other operation to complete nor sacrificing the linearizability of the resulting history. The following theorem captures this characteristic.

Theorem 1 *Let H be any finite linearizable history and $inv[op]$ any pending operation invocation in H. There is a response $r = resp[op]$ such that $H \cdot r$ is linearizable.*

Proof As H is incomplete and linearizable, there is a completion of H, H' that is linearizable, i.e., that has a linearization L. of H. If L contains $inv[op]$ and its matching response r, then L is also linearization of $H \cdot r$. If L contains neither $inv[op]$ not r (i.e., H' does not contain $inv[op]$), then $L' = L \cdot inv[op] \cdot r$ is a linearization of $H' \cdot inv[op] \cdot r$, which means that $H \cdot r$ is linearizable. $\Box_{Theorem\ 2}$

2.4. Composition

We discuss here a fundamental characteristic of linearizability as a *property*, i.e., as a set of histories. A property P is said to be *compositional* if whenever it holds for each of the objects of a set, it holds for the entire set. For each history H, we have $\forall X\ H|X \in P$ if and only if $H \in P$. A compositional property is also said to be *local*. Intuitively, compositionality enables to derive the correctness of a composed system from the correctness of the components. This property is crucial for modularity of programming: a correct (linearizable) compositions can be obtained from correct (linearizable) components.

Theorem 2 *A history H is linearizable if and only if, for each object X involved in H, $H|X$ is linearizable.*

Proof The "only if" direction is a consequence of the definition of linearizability: given that H is linearizable for each object X involved in H, $H|X$ is linearizable. Indeed, for every linearization S of H, $S|X$ is a linearization of $H|X$.

To prove the other direction, consider a history H, where for each object X, $H|X$ has a linearization, denoted S_X, let $\rightarrow_X$ denote the total order in S_X of the operation on X in H. We show below that the relation $\rightarrow = \bigcup_X \{\rightarrow_X\} \cup \{\rightarrow_H\}$ does not induce any cycle. This means that its transitive closure is a partial order, and its linear extension S is a linearization of H.

Assume by contradiction that $\rightarrow$ contains a cycle. Recall that $\rightarrow_X$ and $\rightarrow_H$ are transitive. We can thus replace any fragment of the form $op_1 \rightarrow_X op_2 \rightarrow_X op_3$ (respectively, $op_1 \rightarrow_H op_2 \rightarrow_H op_3$) with $op_1 \rightarrow_X op_3$ (respectively, $op_1 \rightarrow_H op_3$). Moreover, since every operation concerns exactly one object, the cycle cannot contain fragments of the form $op_1 \rightarrow_X op_2 \rightarrow_Y op_3$ for $X \neq Y$. Hence, the cycle alternate edges of the form $\rightarrow_X$ with edges $\rightarrow_H$.

Now consider the fragment $op_1 \rightarrow_H op_2 \rightarrow_X op_3 \rightarrow_H op_4$. Recall that $\rightarrow_X$ is the order of operations in S_X, a linearization of $H|X$. Since S_X respect real time, we have $op_3 \not\rightarrow_X op_2$, i.e., the invocation of op_2 precedes the response of op_3 in $H|X$ (and, thus, in H). Since $op_1 \rightarrow_H op_2$, the response of op_1 precedes the invocation of op_2 and, thus, the response of op_3. Since $op_3 \rightarrow_H op_4$, the response of op_3 and, thus, the response of op_1 precedes the invocation of op_4 in H. Hence, $op_1 \rightarrow_H op_4$, i.e., we can shorten the fragment to one edge $\rightarrow_H$. By eliminating all edges of the form $\rightarrow_X$ we obtain a cycle of edges $\rightarrow_H$—a contradiction with the definition of $\rightarrow_H$ based on the real-time precedence between operations in H that cannot induce cycles.

Hence the transitive closure of $\rightarrow$ is irreflexive and anti-symmetric and, thus, has a linear extension: a total order on operations in H that respects $\rightarrow_H$ and $\rightarrow_X$, for all X. Consider the sequential history S induced by any such total order. Since, for all X, $S|X = S_X$ and S_X is legal, S is legal. Since $\rightarrow_H \subseteq \rightarrow_S$, S respects the real-time order of H. Finally, since each S_X is equivalent to a completion of $H|X$, S is equivalent to a completion of H, where each incomplete operation on an object X is completed in the way it is completed in S_X. Hence, S is a linearization of H. $\Box_{Theorem\ 2}$

The importance of (real) time

Linearizability stipulates correctness with respect to a sequential execution: an operation needs to appear to take effect instantaneously, respecting the sequential specification of the object. In this respect, linearizability is similar to *sequential consistency*, a classical correctness criteria for shared objects. There is however a fundamental difference between linearizability and sequential consistency, and this difference is crucial to making linearizability compositional, which is not the case for sequential consistenty, as we explain below.

Sequential consistency is a relaxation of linearizability. It only requires that the real-time order is preserved if the operations are invoked by the same process, i.e., S is only supposed to respect the *process-order* relation.

More specifically, a history H is *sequentially consistent* if there is a "witness" history S such that:

1. H and S are equivalent,
2. S is sequential and legal.

Both linearizability and sequential consistency require a witness sequential history. However, and as we pointed out, sequential consistency has no further requirement related to the occurrence order of operations issued by different processes (and captured by the real-time order). It is based only on a logical time (the one defined by the witness history). In some sense, with linearizablity, after p_1 has finished its operation en enqueued element a, p_1 could "call" p_2 and inform it about the availability of "a": p_2 will then be sure to find a. Everything happens as if indeed the enqueue of a was executed at a single point in time.

Clearly, any linearizable history is also sequentially consistent. The contrary is not true. A major drawback of sequential consistency is that it is not compositional. To illustrate this, consider the counter-example described in Figure 2.13. The history H depicted in the picture involves two processes p_1 and p_2 accessing two shared registers R_1 and R_2. It is easy to see that the restriction H to each of the registers is sequentially consistent. Indeed, concerning register R_1, we can re-order the read of p_1 before the write of p_2 to obtain a sequential history that respects the semantics of a register (initialized to 0). This is possible because the resuting sequential history does not need to respect the real-time ordering of the operations in the original history. Note that the history restricted to R_1 is not linearizable. As for register R_2, we simply need to order the read of p_1 after the write of p_2.

Nevertheless, the system composed of the two registers R_1 and R_2 is not sequentially consistent. In every legal equivalent to H, the write on R_2 performed by p_2 should precede the read of R_2 performed by p_1: p_1 reads the value written by p_2. If we also want to respect the process-order relation of H on p_1 and p_2, we obtain the following sequential history: $p_2.\mathit{Write}_{R_1}(1)$; $p_2.\mathit{Write}_{R_2}(1)$; $p_1.\mathit{Read}_{R_2}()$ 1; $p_1.\mathit{Read}_{R_1}()$ 0. But the history is not legal: the value read by p_1 in R_1 is not the last written value.

sequential history respecting the process-order relation of H must have

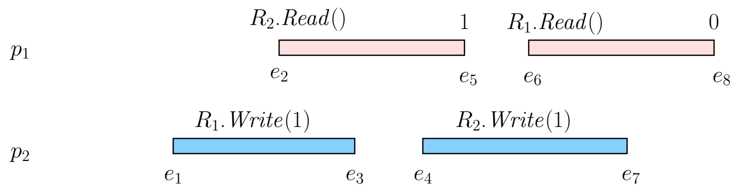


Figure 2.13.: Sequential consistency is not compositional

2.5. Safety

It is convenient to reason about the correctness of a shared object implementation by splitting its properties into *safety* and *liveness*. Intuitively, safety properties ensure that nothing "bad" is ever going to happen whilst liveness properties guarantee that something "good" eventually happens.

More specifically, a *property* is a set of (finite or infinite) histories. Now a property P is a safety property if:

- P is *prefix-closed*: if $H \in P$, then for every prefix H' of H, $H' \in P$.
- P is *limit-closed*: for every infinite sequence $H_0, H_1, \ldots$ of histories, where each H_i is a prefix of H_{i+1} and each $H_i \in P$, the limit history $H = \lim_{i \to \infty} H_i$ is in P.

Knowing that a property is a safety one helps prove it in the following sense. To ensure that a safety property P holds for a given implementation, it is enough to show that every *finite* history is in P: a history is in P if and only if each of its *finite* prefixes is in P. Indeed, every infinite history of an implementation is the limit of some sequence of ever-extending finite histories and thus should also be in P.

Theorem 3 *Linearizability is a safety property.*

The proof of Theorem 3 uses a slight generalization of König's infinity lemma formulated as follows:

Lemma 1 *(König's Lemma) Let G be an infinite directed graph such that (1) each node of G has finite outdegree, (2) each vertex of G is reachable from some root vertex of G (a vertex with zero indegree), and (3) G has only finitely many roots. Then G has an infinite path with no repeated nodes starting from some root.*

Now we prove Theorem 3, i.e., we show that the set of linearizable histories is prefix- and limit-closed. Recall that we only consider objects with finite non-determinism: an operation applied to a given object state may return only finitely many responses and cause only a finite number of state transitions.

Proof Consider a linearizable history H. Since linearizability is compositional, we can simply assume that H is a history of operations on a single (composed) object X. We show first that any H', a prefix of H, is also linearizable (with respect to X).

Let S be any linearization of H, i.e., a sequential legal history that is equivalent to (a completion of H) and respects the real-time order of H. Now we construct a sequential history S' as follows: we take the shortest prefix of S that contains all complete operations of H'. Since S contains all compete operations of H', such a prefix of S exists.

We claim that S' is a linearization of H'. We complete H' by removing operations that do not appear in S' and adding responses to incomplete operations in H' that are present in S'. This way only incomplete operations are removed from H' since, by construction, all operations that are complete in H' appear in S'. Let $\bar{H}'$ denote the resulting complete history.

First we observe that complete histories S' and $\bar{H}'$ consist the same set of operations. By construction, every operation in $\bar{H}'$ appears in S'. Now suppose, by contradiction, that S' contains an operation *op* that does not appear in $\bar{H}'$. Since only operations that do not appear in S' were removed from H' to obtain $\bar{H}'$, *op* does not appear in H' either. Since S' is the shortest prefix of S that contains all complete operations of H, *op* cannot be the last operation appearing in S'. Moreover, for the same reason, the last operation in S' must be complete in H', we denote this operation by *op*$'$. Since *op* does not appear in H' and *op*$'$ is complete in H', we have $op' <_H op$. But *op* precedes *op*$'$ in S' (and, thus, in S), i.e., $op <_S op'$. Hence, S violates the real-time order of H—a contradiction.

Since S' is a prefix of a legal history it is also legal. Moreover, S' and $\bar{H}'$ contain the same set of operations and S' respects the real-time order in $\bar{H}'$: if $<_{\bar{H}'} \subseteq <_{S'}$ (otherwise, S would violate the real-time order in H).

Consider any local history $\bar{H}'|p_i$. Recall that we only assume well-formed histories and, thus, $\bar{H}'|p_i$ is sequential. Since S' and $\bar{H}'$ contain the same set of operations and S' respects the real-time order of $\bar{H}'$, we have $S'|p_i = \bar{H}'|p_i$. Hence, S' and $\bar{H}'$ are equivalent.

Thus, S' is indeed a linearization of H' and, thus, linearizability is prefix-closed.

To show that linearizability is limit-closed, we consider an infinite sequence of ever-extending linearizable histories $H_0, H_1, H_2, \ldots$. Our goal is to show that $H = \lim_{i\to\infty} H_i$ is linearizable. We assume that H_0 is the empty history and each H_{i+1} is a one-event extension of H_i (by prefix-closedness, prefix of every H_i is linearizable, so we do not lose generality this way).

Now we construct a directed graph $G = (V, E)$ as follows. Vertices of G are all tuples (H_i, S, Q), where $i = 0, 1, \ldots, |H|$, S is any linearization of H_i that ends with a *complete* operation present in H_i, and Q is any sequence of object states that witnesses the legality of H. Now there is an directed edge $((H_i, S, Q), (H_j, S', Q')$ in G if and only if $j = i + 1$, S is a prefix of S' and Q is a prefix of Q'.

Note that each H_i has at least one vertex (H_i, S, Q). Indeed, by taking any linearization of H_i and removing operations at the end of it that are incomplete in H_i, we obtain a linearization of a completion of H_i in which these operations are removed. Thus, there exists a linearization S of H_i that ends with a complete operation in H_i. Since S is legal, it must have a witness sequence of states Q.

We use König's lemma to show that the resulting graph G contains an infinite path $(H_0, S_0), (H_1, S_1), \ldots$ and the limit $\lim_{i\to\infty} S_i$ is a linearization of the infinite limit history H.

First we observe that each non-empty vertex (H_{i+1}, S', Q') is connected to some (H_i, S, Q). There are two cases to consider:

- The last operation *op* of S' is a complete operation in H_i. In this case, S' is also a linearization of H_i. Indeed, even if the last event of H_{i+1} is the invocation of a new operation *op*$'$, this operation cannot appear in S': it can only appear before *op* in S' violating the real-time order in H_{i+1}. Thus, (H_i, S', Q') is a vertex in G.

- The last operation *op* of S' is not a complete operation in H_i. Recall that S' ends with an operation *op* that is complete in H_{i+1} and H_{i+1} extends H_i with one event only. Thus, the last event of H_{i+1} is the response of *op*. Thus, H_i and H_{i+1} contain the same set of operations, except that *op* is incomplete in H_i. Let S be the longest prefix of S' that ends with a complete operation in H_i. Since S' is legal, S is also legal. By construction, every complete operation in H_i appears in S and no operation appears in S if it does not appear in H_i. Thus, S is a linearization of H_i and (H_i, S, Q), where Q is the prefix of Q' that witnesses the legality of S, is a vertex in G.

Inductively, we derive that each vertex (H_i, S, Q) is reachable from vertex (H_0, S_0, Q_0), where H_0, S_0 and W_0 are empty sequences. The only *root vertex* of G (a vertex that has no incoming edges) is thus (H_0, S_0, W_0).

Now we show that the outdegree of every vertex of G is finite. There are only finitely many operations in H_{i+1} and each linearization of H_{i+1} is a permutation of these operations. Thus, since we only consider objects with finite non-determinism, there can only be finitely many vertices of the form (H_{i+1}, S', Q'). Since all outgoing edges of any vertex (H_i, S, Q) are directed to vertices of the form (H_{i+1}, S', Q'), the outdegree of every such vertex is also finite.

By König's lemma, G contains an infinite path starting from the root vertex: $(H_0, S_0, Q_0), (H_1, S_1, Q_1), \ldots$. We argue now that the limit $S = \lim_{i\to\infty} S_i$ is a linearization of the infinite limit history H. By construction, S respects the real-time order of H, otherwise there would be a vertex (H_i, S_i, Q_i) such that S_i is not equivalent to H_i or violates the real-time order of H_i. Also, S contains all complete operations of H and, thus, S is equivalent to a completion of H. S is also legal since each of its prefixes is legal. Thus, S is indeed a linearization of H, which concludes the proof that linearizability is a safety property.

$\square_{Theorem\ 3}$

Thus, the set of linearizable histories is indeed prefix-closed and limit-closed, so in the rest of this book, we only consider finite histories in the proofs of linearizability.

2.6. Summary

This chapter studies the meaning of the notion of a correct object implementation. Namely, to be correct, all histories generated by the object implementation need to be linearizable. The responses returned by the object in a concurrent history are those that could have been returned by the object if accessed sequentially. Proving this typically boils down to determining a linearization point for each operation in the given history.

Linearizability has some important characteristics. First, it reduces the difficult problem of reasoning about a concurrent system into the problem of reasoning about a sequential one. We simpy need a sequential specification of an object to reason about the correctness of a system made of processes concurrently accessing that object. Linearizabiliy is also compositional. It is enough to prove that each object in a set (of objects) is linearizable to conclude that the system composed of the set is linearizable. Linearizability is also non-blocking, which basically means that ensuring it never forces processes to wait for each other.

As pointed out however, linearizability is only a partial answer to the question of correctness. It does say what response should be forbidden to be returned by an object but does not say when the object should actually return some response. In fact, and as we will see in the next chapter, to be considered correct, the object implementation should not only be linearizable but should also be *wait-free*. Whilst linearizability covers safety, wait-freedom covers liveness.

2.7. Bibliographic notes

The notion of sequential consistency has been introduced by Lamport [47]. Linearizability was initially studied, under the name *atomicity*, in the context of atomic read/write objects (registers) by Lamport [49] and Misra [57]. The notion of sequential specification of a type was introduced by Weihl in [74]. The generalization of linearizability to any object type has been developed by Herlihy and Wing [40].

The concepts of safety and liveness were introduced by Lamport [46] and refined by Alpern and Schneider [3], originally defined for infinite histories only. Lynch reformulated the notions for finite histories and proved that linearizability, when applied to deterministic objects is a safety property [56]. Guerraoui and Ruppert [31] showed that linearizability is not limit-closed if objects can expose infinite non-determinism. In other words, linearizability is not a safety property for objects with unbounded non-determinism.

3. Progress

3.1. Introduction

The previous chapter focused on the property of *linearizability*, which basically precludes concurrent operations that do not appear as if executed sequentially. Linearizability (when applied to objects with finite non-determinism) is a *safety* property: it states what *should not* happen in an execution.

Such a property is in fact easy to satisfy. Think of an implementation (of some shared object) that simply never returns any response. Since no operation would ever complete, the history would basically be empty and would be trivial to linearize: no response, no need for a linearization point. But this implementation would be useless. In fact, to prevent such implementations, we need some *progress* property stipulating that certain responses *should* appear in a history, at least eventually and under certain conditions. Ideally, we would like every invoked operation to eventually return a matching response. But this is impossible to guarantee if the process invoking the operation crashes, e.g., the process is paged out by the operating system which could decide not to schedule that process anymore.

Nevertheless, one might require that a response is returned to a process that is scheduled by the operating system to execute enough *steps* of the algorithm implementing that operation (i.e., implementing the object exporting the operation). As we will see below, a step here is the access to a low-level object (used in the implementation) during the operation's execution.

To express such requirement more precisely, we need to carefully define the notion of object *implementation* and zoom into the way processes execute the algorithm implementing the object, in particular how their steps are scheduled by the operating system.

In the following, we introduce the notion of *implementation history*: this is a *lower level* notion than the history notion presented in the previous chapter and which describes the interaction between the processes and the object being implemented (*high-level history*) The concept of low-level history will be used to introduce progress properties of shared object implementations.

3.2. Implementation

In order to reason about the very notion of implementation, we need to distinguish the very notions of *high-level* and *low-level* objects.

3.2.1. High-level and low-level objects

To distinguish the shared object to be implemented from the underlying objects used in the implementation, we typically talk about a *high-level* object and underlying *low-level* objects. (The latter are sometimes also called *base* objects and the operations they export are called *primitives*). That is, a process invokes *operations* on a high-level object and the implementation of these operations requires the process to invoke *primitives* of the underlying low-level (base) objects. When a process invokes such a primitive, we say that the process performs a *step*.

The very notions of "high-level" and "low-level" are relative and depend on the actual implementation. An object might be considered high-level in a given implementation and low-level in another one. The object to be implemented is the high-level one and the objects used in the implementation are the low-level ones. The low-level objects might capture basic synchronization constructs provided in hardware and in this case the high-level ones are those we want to emulate in software (the notion of *emulation* is what we call *implement*). Such emulations are motivated by the desire to facilitate the programming of concurrent applications, i.e. to provide the programmer with powerful synchronization abstractions encapsulated by high-level objects. Another motivation is to reuse programs initially devised with the high-level object in mind in a system that does not provide such an object in hardware. Indeed, multiprocessor machines do not all provide the same basic synchronization abstractions.

Of course, an object O that is low-level in a given implementation A does not necessarily correspond to a hardware synchronization construct. Sometimes, this object O has itself been obtained from a software implementation B from some other lower objects. So O is indeed low-level in A and high-level in B. Also, sometimes the low-level objects are assumed to be linearizable, and sometimes not. In fact, we will even study implementations of objects that are not linearizable, as an intermediate way to build linearizable ones.

3.2.2. Zooming into histories

So far, we represent computations using histories, as sequences of events, each representing an invocation or a response on the object to be implemented, i.e, the high-level object.

Implementation history. In contrast, reasoning about progress properties requires to zoom into the invocations and responses of the lower level objects of the implementations, on top of which the high-level object is built. Without such zooming we may not be able to distinguish a process that crashes right after invoking a high-level object operation and stops invoking low-level objects, from one that keeps executing the algorithm implementing that operation and invoking primitives on low-level objects. As we pointed out, we might want to require that the latter completes the operation by obtaining a matching response, but we cannot expect any such thing for the former. In this chapter, we will consider as a *implementation history*, the low-level history involving invocations and responses of low-level objects. This is a refinement of the higher level history involving only the invocations and responses of the high-level object to be implemented.

Consider the example of a fetch-and-increment (counter) high-level-object implementation, as we describe it below in Section 3.4.1. As low-level objects, the implementation uses an infinite array $T[,\dots,\infty]$ of TAS (test-and-set) objects and a snapshot-memory object *my-inc*. The high-level history here is a sequence of invocation and response events of *fetch-and-increment* operations, while the low-level history (or implementation history) is a sequence of primitive events *read*(), *update*(), *snapshot*() and *test-and-set*().

The two faces of a process. To better understand the very notion of a low-level history, it is important to distinguish the two roles of a process. On the one hand, a process has the role of a *client* that sequentially invokes operations on the high-level object and receives responses. On the other hand, the process also acts as a *server* implementing the operations. While doing so, the process invokes primitives on lower level objects in order to obtain a response to the high-level invocation.

It might be convenient to think of the two roles of a process as executed by different entities and written by two different programmers. As a client, a process invokes object operations but does not control the way the low-level primitives implementing these operations are executed. The programmer writing this part does typically not know how an object operation is implemented. As a server, a process executes the implementation algorithm made up of invocations of low-level object primitives. This algorithm is typically written by a different programmer who does not need to know what client applications will be using this object. Similarly, the client application does not need to know how the objects used are implemented, except that they ensure linearizability and some progress property as discuss below.

Scheduling and asynchrony. The execution of a low-level object operation is called a *step*. The interleaving of steps in an implementation is specified by a *scheduler* (itself part of an operating system). This is outside of the control of processes and, in our context, it is convenient to think of a scheduler as an *adversary*. This is because, when devising an algorithm to implement some high-level object, one has to cope with worst-case strategies the scheduler may choose to defeat the algorithm. This is then viewed as an adversarial behavior.

A process is said to be *correct* in a low-level history if it executes an infinite number of steps, i.e., when the scheduler allocates infinitely many steps of that process. This "infinity" notion models the fact that the process executes as many steps as needed by the implementation until all responses are returned. Otherwise, if the process only takes finitely many steps, it is said to be *faulty*. In this book, we only assume that faulty processes *crash*, i.e., permanently stop performing steps, otherwise they never deviate from the algorithm assigned to them. In other words, they are not malicious (we also say they are not *Byzantine*).

Unless explicitly stated otherwise, the system is assumed to be *asynchronous* , i.e., the relative speeds of the processes are unbounded: for all $\Phi \in \mathbb{N}$ and processes p and q, there is an execution in which p takes Φ steps while process q takes only one step. Basically, an asynchronous system is controlled by a very weak scheduler, i.e., a scheduler that may prevent a correct process from taking steps for an arbitrary (but finite) periods of time.

3.3. Progress properties

As pointed out above, a trivial way to ensure linearizability would be to do nothing, i.e., return no response to any operation invocation. This would preclude any history that violates linearizability by simply precluding any history with a response.

Besides this (clearly, meaningless) approach, a popular way to ensure linearizability is to use *critical sections* (say using *locks*), preventing concurrent accesses to the same high-level shared object. In the simplest case, every operation on a shared object is executed as a critical section. When a process invokes an operation on an object, it first requests the corresponding lock, and the algorithm of the operation is executed by the process only when the lock is acquired. If the lock is not available, the process waits until the lock is released. After a process obtains the response to an operation, it releases the corresponding lock. This approach also trivially ensures linearizability because the linearization points of the operations of a history correspond to the moment at which the lock is acquired for the operation.

As we discussed in Chapter 1, such an implementation of a shared object has an inherent drawback: the crash of a process holding the lock on an object prevents any other process from completing its operation. In practice, the process holding the lock might be preempted for a long period of time, while all processes contending on the same object remain blocked. When processes are asynchronous (i.e., the scheduler can arbitrarily preempt processes) which is the default assumption we consider, there is no way for a process to know whether another process has crashed (or was preempted for a long while) or is only very slow. In a system with a couple of processors, this might not be considered a big deal. But in a modern architecture with a very large number of processors, having a single point of blocking might be considered unacceptable.

This book focuses on *robust* shared object implementations with progress properties precluding situations where the crash of some strict subset of processes prevents every other process from making progress. This models the requirement that processes that are delayed by the operating system should not block all other processes from progressing. Hence, we preclude the use of critical sections or locks.

- Informally, we say that an implementation is *lock-based* if it allows for a situation in which some process running in isolation after some finite execution is never able to complete its operation.
- Taking a negation of this property, we state that an implementation *does-not-employ-locks* if starting after any finite execution, every process can complete its operation in a finite number of its own steps.

Intuitively, this property, called *obstruction-freedom* (or *solo termination*), must be satisfied by any implementation where the crash of any process does not prevent other processes from making progress. Below we discuss this property in more details together with some of its variants.

3.3.1. Variations

Several progress properties preclude the usage of locks:

- Obstruction-freedom (also called *solo termination*). An implementation (of a shared object) is obstruction-free, if any of its operations returns a response if it is eventually executed without concurrency by a correct process.

  The operation is said to be *eventually executed without concurrency* if there is a time after which the only process to take *step* involving the object is the process that invoked that operation.[1]
- Non-blockingness (*partial termination*). This property, strictly stronger than obstruction-freedom, states that at least one of several correct processes executing operations on the same object, terminates its operation. Intuitively, non-blockingness can be interpreted as *deadlock-freedom* (despite asynchrony and crashes).
- Wait-freedom (also called *global termination*). This property is even stronger. It states that any correct process that executes an operation eventually returns a response. Wait-freedom can be viewed as *starvation-freedom* (despite asynchrony and crashes).

3.3.2. Bounded termination

Wait-freedom, the strongest of the properties above, does not stipulate any bound on the number of steps that a process needs to execute before obtaining a matching response for the high-level object operation it invoked. Typically, this number of steps can depend on the behavior of the other processes. It could be small if no other process performs any step, and gets bigger when all processes perform steps (or the opposite), while remaining always finite, regardless of the number and timing of crashes.

- An implementation is *bounded wait-free* if there exists a bound $B \in \mathbb{N}$ such that every process p that invokes an operation receives a matching response within B of its own (not necessarily consecutive in the execution) steps.

  In other words, there is no prefix of a low-level history in which a process invokes an operation and executes B steps without obtaining a matching response.

[1] There is an alternative, weaker notion of contention, called *interval* contention. An operation encounters interval contention if it overlaps with another operation (this does not need to take steps). Step contention implies interval contention, but not vice versa. However, an alternative definition of obstruction-freedom requiring that an operation returns if it runs in the absence of interval contention does not preclude the usage of locks. An operation grabs the lock on the shared object, executes the operation on the object, and releases the lock before returning the response.

Showing that an implementation is bounded wait-free consists in exhibiting an upper bound on the number of steps needed to return from any operation. That upper bound is usually defined by a function of the number n of processes (e.g., $O(n^2)$). One can similarly define notions like bounded solo termination or bounded partial termination.

3.3.3. Liveness

Recall that safety properties (Section 2.5) are used to declare what it means for an implementation to reach an undesired state. To show that an implementation satisfies a safety property P, it is sufficient to check if each of its *finite* executions satisfies P.

In contrast, a *liveness* property ensures that the implementation eventually reaches some desired state. More specifically, we say that P is a liveness property if *any* finite execution has an extension in P. Hence, no matter what state our implementation is in, there is always a chance to reach a desired state in some extension of the current execution. To show that an implementation satisfies a liveness property P, we should thus show that all its infinite executions are in P.

Interestingly, every property can be represented as an intersection of a safety property and a liveness property [56]. Linearizability is a safety property (Section 2.5). Wait-freedom, as we can easily see, is a liveness property. Indeed, we can only violate wait-freedom in an infinite execution: every finite execution in which an operation invoked by a given process has an extension in which the operation returns. Similarly, non-blockingness and obstruction-freedom are also liveness properties. For example, the only way to violate obstruction-freedom is to exhibit an execution in which a process takes infinitely many steps without completing an invoked operation.

It is interesting to notice that *bounded wait-freedom* is, in fact, a safety property. Indeed, B-bounded wait-freedom is violated in a finite execution where an operation does not return after B steps of the process that invoked it. It is not difficult to see that B-bounded wait-freedom is prefix-closed and limit-closed. Therefore, to prove that an implementation is, e.g., linearizable and B-bounded wait-free, it is enough to consider its finite executions.

3.4. Linearizability and wait-freedom

3.4.1. A simple example

The algorithm described in Figure 3.1 is a simple wait-free linearizable implementation of a *fetch-and-increment (FAI* object using an infinite array of *test-and-set TAS* objects $T[1, \ldots, \infty]$ and a *snapshot memory* object *My_inc*.

- The high-level object is the FAI. This object stores an integer value and exports one operation *fetch-and-increment*(). The sequential specification of this operation basically increments the value of the integer value and returns the previous value.

- The low-level objects used in the implementation include TAS objects. Each of these exports one (primitive) operation *test-and-set*() that returns 0 or 1. The sequential specification of this operation guarantees that the first invocation of *test-and-set*() on the object returns 1 and all subsequent invocations return 0. Intuitively, a TAS object allows a single process to distinguish itself from the rest of the processes. Such objects are typically provided by many multi-core machines.

- The snapshot memory is also a low-level object used in the implementation. It can be seen as an array of n registers, one for each process, such that each process p_i can atomically write a value v

to its dedicated register with an operation *update*(i, v) and atomically read the content of the array using an operation *snapshot*$()$. [2]

```
Shared
    $T[1,\dots,\infty]$: $n$-process TAS objects
    $My\_inc[1,\dots,\infty]$: snapshot memory, initialized to 0

Local
    $entry$, $c$ (initially 0), $S$

operation $fetch\text{-}and\text{-}increment()$:
    $c \leftarrow c + 1$;
    $My\_inc.update(i, c)$;
    $S \leftarrow My\_inc.snapshot()$;
    $entry \leftarrow sum(S)$;
    while $T[entry].test\text{-}and\text{-}set() \neq 0$ do
        $entry \leftarrow entry - 1$;
    $return(entry - 1)$
```

Figure 3.1.: Fetch-and-increment implementation: code for process p_i

The algorithm in Figure 3.1 depicts the code executed by every process p_i of the system. It works as follows. To increment the value of the FAI object (i.e., to execute a *fetch-and-increment*$()$ operation), p_i first increments its dedicated register in the snapshot memory *My_inc*. Then p_i takes a snapshot of the memory and evaluates *entry* as the sum of all its elements. Then, starting from the $T[entry]$ down to 1, p_i invokes operations *test-and-set*$()$ until some TAS object returns 1. The index of this TAS object minus 1 is then returned by *fetch-and-increment*$()$ operation.

Intuitively, when p_i evaluates its local variable *entry* to ℓ, at most ℓ processes have previously incremented their positions and, thus, at least one TAS object in the array $T[1,\dots,\ell]$ is "reserved" for p_i (p_i is one of these ℓ processes). Every process that increments its position in *My_inc* later will obtain a strictly higher value of *entry*. Thus, eventually, every operation obtains 1 from one of the TAS objects and returns. Moreover, since a TAS object returns 1 to exactly one process, every returned value is unique.

Notice that the number of steps performed by a *fetch-and-increment*$()$ operation is finite but in general unbounded (the implementation is not bounded wait-free). This is because an unbounded number of increments can be performed by other processes in the time lag between a process p_i increments it position in *My_inc* and the moment p_i takes a snapshot of *My_inc*. It is however not difficult to modify the algorithm so that every operation performs $O(n^2)$ steps.

3.4.2. A more sophisticated example

Proving that a given implementation satisfies linearizability and wait-freedom can be extremely tricky sometimes. To illustrate this, consider now the algorithm of Figure 3.2 that intends to implement an unbounded FIFO queue. The sequential specification of this object has been given in Section 2.1 of Chapter 2.

The algorithm is quite simple. The system we consider here is made up of producers (clients) and consumers (servers) that cooperate through an unbounded FIFO queue. A producer process repeats forever the following two statements:

[2] In Chapter 8, we show how snapshot memory can be implemented in a wait-free and linearizable manner using only read-write registers.

1. Prepare a new item v;
2. Invoke the operation $Enq(v)$ to deposits v in the queue.

Similarly, a consumer process repeats forever the following two statements:

1. Withdraw an item from the queue by invoking the operation $Deq()$
2. Consume that item.

If the queue is empty, then the default value *nil* is returned to the invoking process. (This default value that cannot be deposited by a producer process.) We assume that no processing by the consumer is associated with the *nil* value.

The algorithm depicted in Figure 3.2 relies on an unbounded array $Q[0, \ldots, \infty]$, where entry of the array is initialized to *nil* and is used to store the items of the queue. Also, the implementation uses a shared variable *NEXT* (initialized to 1) as a pointer to the next available slot of the array Q for a new value to be deposited.

To enqueue an item to the queue, the producer first locates the index of the next empty slot in the array Q, reserves it, and then stores the item in that slot. To dequeue a value, the consumer first determines the last entry of the array Q that has been reserved by a producer. Then, it reads the elements of the array Q in ascending order until it finds an item different from the default value *nil*. If it finds one, it returns it. Otherwise, the default value is returned.

The variable *NEXT* is provided with two primitives denoted read() and fetch&add(). The invocation *NEXT*.fetch&add(x) returns the value of *NEXT* before the invocation and adds x to *NEXT*. Similarly, each entry $Q[i]$ of the the array is provided with two primitives denoted write() and swap(). The invocation $Q[i]$.swap(v) writes v in $Q[i]$ and returns the value of $Q[i]$ before the invocation.

The execution of the read(), write(), fetch&add() and swap() primitives on the shared base objects (*NEXT* and each variable $Q[i]$) are assumed to be linearizable. The primitives read() and write() are implicit in the code of Figure 3.2 (they are in the assignment statements denoted "←").

The algorithm does not use locks: no process can block other processes forever. Furthermore, each value deposited in the array by a producer will be withdrawn by a swap() operation issued by a consumer (assuming that at least one consumer is correct).

```
operation $Enq(v)$:
    $in \leftarrow$ NEXT.fetch&add (1);
    $Q[in] \leftarrow v$;
    return ()

operation $Deq()$:
    $last \leftarrow NEXT - 1$;
    for $i$ from 0 until $last$ do
        $aux \leftarrow Q[i]$.swap (nil);
        if $(aux \neq \perp)$ then return $(aux)$
    return (nil)
```

Figure 3.2.: Enqueue and dequeue implementations

It is easy to see that the implementation is wait-free: every process completes each of its operations in a finite number of its own steps: the number of steps performed by *Enq*() is two, and the number of steps performed by *Deq*() is proportional to the queue size as evaluated in the first line of its pseudocode.

But is the implementation linearizable? Superficially, yes: if no dequeue operation returns *nil*, we can order operations based on the times when the corresponding updates of $Q[]$ (a write performed by *Enq*$()$ or a successful swap performed by *Deq*$()$) takes place.

However, if a dequeue operation returns *nil* it is not always possible to find the right place for it in a legal linearization. Consider for instance the following scenario:

1. Process p_1 performs *Enq*(x). As a result, the value of *NEXT* is 1, and $Q[0]$ stores x.
2. Process p_2 starts executing *Deq*$()$ and reads 1 in *NEXT*.
3. Process p_1 performs *Enq*(y). The value of *NEXT* is now 2, $Q[0]$ stores x, and $Q[1]$ stores y.
4. Process p_3 performs *Deq*$()$, reads 2 in *NEXT*, finds x in $Q[0]$ and returns x. The value of $Q[0]$ is *nil* now.
5. Finally, p_2 reads $\perp$ in $Q[0]$ and completes *Deq*$()$ by returning *nil*.

In this execution: we have the following partial order on operations: p_1.*Enq*$(x) \to p_1$.*Enq*$(y) \to p_3$.*Deq*(x), and p_1.*Enq*$(x) \to p_2$.*Deq(nil)*. Thus, there are only three possible ways to linearize p_2.*Deq(nil)*(: right after p_1.*Enq*(x), right after p_1.*Enq*(y) or right after p_3.*Deq*$()$. In all three possible linearizations, the queue is not empty when p_2 invokes *Deq*$()$, and thus *nil* cannot be returned.

How to fix this problem? One solution is to sacrifice linearizability and not consider operations returning *nil* in a linearization.

Another solution is to sacrifice wait-freedom and instead of returning *nil* in the last line of the *Deq*$()$, repeat the same procedure (evaluating *NEXT* and going through the first *NEXT* elements in $Q[]$) over and over until a non-$\perp$ value is found in $Q[]$. As long as a producer keeps adding items to the queue, every *Deq*$()$ operation is guaranteed to eventually return.

3.5. Summary

To reason about correctness of an object implementation, it is common to consider linearizability, as well as some companion progress property. In this chapter, we studied three progress properties: solo-termination (obstruction-freedom), partial-termination (non-blockingness) and global termination (wait-freedom). All of these are liveness properties, precluding the usage of locks. The first of these properties says that a process that eventually accesses an object alone (with no contention) will get responses when invoking the object's operation. The second property requires a response to be returned to at least one of the correct processes even if there is contention. The last property, wait-freedom, is the strongest. Responses should be returned to every correct process that invokes an operation, i.e., that keeps executing low-level steps. In Chapter **??**, we express other conditions on the executions in which progress must be ensured in the form of generic *adversaries*.

Bibliographic notes

The notion of wait-freedom originated in the work of Lamport [45]. An associated theory was developed by Herlihy [34].

The notion of solo-termination was presented implicitly in [25]. It has been introduced as a progress property in [37] under the name *obstruction-free* synchronization, and then formalized in [9]. More developments on obstruction-freedom can be found in [26]. The minimal knowledge on process failures needed to transform any solo-terminating implementation into a wait-free one was investigated in [30].

Other progress conditions, including those that can be implemented with locks, are discussed in [38]. A systematic perspective on progress conditions is presented in [39].

The algorithms in Figure 3.1 and Figure 3.2 were proposed by Afek et al. [2]. A blocking variant of the algorithm of Figure 3.2 in which *nil* is never returned was given and proved correct by Herlihy and Wing [40].

3.6. Exercises

1. Prove that bounded wait-freedom is a safety property.
2. Show that the algorithm sketched in the last paragraph of Section 3.4.2 indeed violates wait-freedom.

Part II.

Read-write objects

4. Simple register algorithms

The simplest objects that are usually considered in concurrent computing are *registers*, namely shared *storage* objects that export two basic operations: *read* and *write*. For presentation simplicity, and without loss of generality, we focus only on registers that contain integers.

In the following, we shall describe how to *wait-free* implement registers that are *atomic* using registers that are not. We will not directly try to obtain atomic registers but rather proceed incrementally through several steps. At each step, which we believe is interesting in its own right, we will build registers that ensure stronger semantics than the weaker ones we use. The picture to have in mind here is that the weak registers are provided in hardware whereas the stronger ones, implemented on top of the weaker ones, are emulated in software.

4.1. Definitions

Different kinds of registers are usually considered, depending on:

(a) Their *value range*, namely the range of values that can be stored in the register. We typically consider, on the one hand, registers that can contain only binary values, i.e., only 0 or 1, also called *binary* registers, or *bits*, and, on the other hand, registers that contain any value from an a larger set, also called *multi-valued* registers. A multi-valued register can be *bounded* or *unbounded*. A *bounded* register is one whose value range contains exactly b distinct values, e.g., the values from 0 until $b-1$ where b is typically a constant integer by the processes. Otherwise the register is said to be *unbounded*. A register that can contain b distinct values is said to be *b-valued*.

(b) Their *access pattern*, i.e., the number of processes that can read (resp., write in) the register. We distinguish 1-writer 1-reader and multi-writer multi-reader. It is important to notice that we do not consider dynamic access patterns that change over time, i.e., the patter is determined once and for all at the creation of the register. A register is called *single-writer*, denoted 1W (resp., *single-reader*, denoted 1R) if only one specific process, known in advance, and called the *writer* (resp., the *reader*) can invoke a write (resp., read) operation on the register. A register that can be written (resp., read) by multiple processes is called a *multi-writer* (resp., *multi-reader*) register. Such a register is denoted MW (resp., MR). For instance, a binary 1WMR register is a register that (a) can contain only 0 or 1, (b) can be read by all the processes but (c) written by a single process.

(c) Their *concurrent behavior*, i.e., the correctness guarantees ensured when the register is accessed concurrently. Registers that ensure linearizability are sometimes called *atomic* or *linearizable* registers. But, as pointed out earlier, and as we will discuss below, there are interesting forms of registers that provide weaker correctness guarantees. We will consider two such forms, called *safe* and *regular* registers, respectively.

The concurrent behavior of a register. When accessed sequentially, the behavior of a register is simple to define: a read invocation returns the last value written. When accessed concurrently, three main variants have mainly been considered. We overview them below.

Safety A read that is not concurrent with a write returns the last written value. This is the only property ensured by a *safe* register, which boils down to saying that the register does not provide any guarantee if accessed concurrently. Such a register supports only a single writer. If this writer is concurrent with a read, this read can return any value in the range domain of the register, including a value that has never been written. A binary safe register is thus a bit flickering under concurrency.

Regularity A *regular* register ensures, together with the safety property above, that a read that is concurrent with a write returns the value written by that write or the value written by the last preceding write. A regular register also only supports a single writer.

It is important to notice that such a register can, if two consecutive (non-overlapping) reads are concurrent with a write, returns the value being written (the new value) and then returns later the previous value written (the old value). This situation is called the *new/old inversion*. It could occur even if the two reads are issued by the same process, as depicted on Figure 4.1. More generally, a read that overlaps *several* write operations can return the value written by any of these writes as well as the value of the register before these writes.

Atomicity An *atomic* (*linearizable*) register is one that ensures linearizability. Such a register ensures, in addition to the safety and regularity properties above, that a *new/old inversion* never happens. The second read must return the same or a "newer" value. Basically, considering Figure 4.1, if the first read of p_1 returns 1, then the second read of p_1 has to return 1.

The *weakest* kind of registers is one that can only store one bit of information (binary), can be read by a single process and written by a single process, while not ensuring any guarantee on the value returned by a read that is concurrent by a write. On the other hand, the *strongest* kind of register is the MWMW multi-valued atomic register.

An algorithm that implements a register of a given kind from a register of a weaker kind is sometimes called register *transformation* or *reduction*, the former (high-level) register being "reduced" to the latter one, used as a base object in the implementation. We also say that the high-level register is *emulated by*, or *constructed from*, the lower-level one.

Before presenting several register transformations, we will highlight first some fundamental techniques that enable to argue about the correctness of a given transformation.

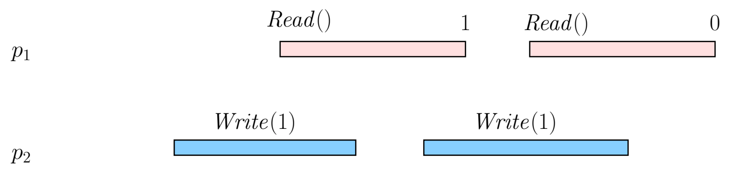


Figure 4.1.: New/old inversion

4.2. Proving register properties

Proving that a register is safe boils down to focusing on the sequential case and ensuring that a read returns the last value written. Proving that a register is regular (or atomic) is more challenging. The very notion of a *reading function* is in this context convenient.

Basically, a reading function is associated with a history. It maps, in that history, every read operation r invoked on the register to some write invocation w of that register. In short, w is the operation that wrote the value returned by r. Without loss of generality, we assume that every history starts with an operation $w(x_0)$ that writes the initial value x_0. No other operation is concurrent with that initial write operation.

We say that a reading function π associated with a history H is *regular* if it satisfies the following two properties:

$A1$: $\forall\, r$: $\neg(r \rightarrow_H \pi(r))$. *(No read returns a value not written yet.)*

$A2$: $\forall\, r, w$ in H: $(w \rightarrow_H r) \Rightarrow \big(\pi(r) = w \;\vee\; w \rightarrow_H \pi(r)\big)$. *(No read returns a value overwitten.)*

We say that a reading function is *atomic* if, besides being regular ($A1$ and $A2$), it satisfies the following property:

$A3$: $\forall\, r1, r2$: $(r1 \rightarrow_H r2) \Rightarrow \big(\pi(r1) = \pi(r2) \;\vee\; \pi(r1) \rightarrow_H \pi(r2)\big)$. *(No new/old inversion.)*

Theorem 4 *H is a history of a 1WMR regular register if and only if H has a regular reading function π.*

Proof Consider H a history of a regular register. We define its reading function π as follows. For any r, a read operation in H that returns x, we define $\pi(r)$ as the last write operation $w(x)$ in H such that $\neg(r \rightarrow_H w(x)$. Since by the definition of a regular register, x is the argument of the latest preceding write or a concurrent write, π satisfies properties $A1$ and $A2$ above.

Now suppose H has a regular reading function. Let r be a complete read operation in H that returns x. Then there exists a write $w(x)$ in H that either (a.1) precedes or (a.2) is concurrent with r in H ($A1$) and (b) is not followed by any write that precedes r in H ($A2$). Thus, r returns either the last written value or a concurrently written value. $\square_{Theorem\ 4}$

Theorem 5 *H is a history of an atomic 1WMR register if and only if H has an atomic reading function π.*

Proof Given a linearizable history H, we construct an atomic reading function as follows. Consider any linearization of H, S, and define $\pi(r)$ as the last write that precedes r in S. By construction, $\pi(r)$ satisfies properties $A1$, $A2$ and $A3$ above.

Now assume H has an atomic reading function π. We use π to construct S, a linearization of H, as follows. We first construct S as the sequence of all writes that took place in H in the order they appear. Since there is only one writer, the writes are totally ordered. If the last write is incomplete, we add to S its complete version. Then we serialize every complete operation r immediately after $\pi(r)$, in such a way that:

$$\textit{if } \pi(r1) = \pi(r2) \textit{ and } r1 \rightarrow_H r2, \textit{ then } r1 \rightarrow_S r2.$$

Clearly the reading function guarantees that $\pi(r)$ writes the value read by r, and thus every read in S returns the last written value (S is legal).

To show that $\rightarrow_H \subseteq \rightarrow_S$, we distinguish the following four possible cases, where $w1$ and $w2$ denote write operations, whereas $r1$ and $r2$ denote read operations.

- $w1 \rightarrow_H w2$. Since S preserves the real-time occurrence order of writes in H, we have $w1 \rightarrow_S w2$.

- $r1 \rightarrow_H r2$. By $A3$, we have $\pi(r1) = \pi(r2)$ or $\pi(r1) \rightarrow_H \pi(r2)$.

  If $\pi(r1) = \pi(r2)$, as $r1$ precedes $r2$ in H, the way S is constructed implies that $r1$ is ordered before $r2$ in S and hence $r1 \rightarrow_S r2$.

  If $\pi(r1) \rightarrow_H \pi(r2)$, then, since S preserves the real-time occurrence order of writes in H and $r1$ and $r2$ are placed just after $\pi(r1)$ and $\pi(r2)$, respectively, in S, we have $r1 \rightarrow_S r2$.

- $r1 \rightarrow_H w2$. By $A1$, either $\pi(r1)$ is concurrent with $r1$ or $\pi(r1) \rightarrow_H r1$. Since $r1 \rightarrow_H w2$ and all writes are totally ordered, we have $\pi(r1) \rightarrow_H w2$. By construction of S, since $\pi(r1)$ is the last write preceding $r1$ in S, $r1 \rightarrow_S w2$.

- $w1 \rightarrow_H r2$. By $A1$ we have $\pi(r2) = w1$ or $w1 \rightarrow_H \pi(r2)$.

  Assume $\pi(r2) = w1$. As $r2$ is serialized just after $\pi(r2)$ in S, we have $\pi(r2) = w1 \rightarrow_S r2$.

  Assume $w1 \rightarrow_H \pi(r2)$. Again, by the way S is constructed, we have $w1 \rightarrow_H \pi(r2) \Rightarrow w1 \rightarrow_S \pi(r2)$. Further, $\pi(r2) \rightarrow_S r2$ ($r2$ is ordered just after $\pi(r2)$ in S), we obtain (by transitivity of $\rightarrow_S$) $w1 \rightarrow_S r2$.

Finally, since S contains all complete operations of H and preserves $\rightarrow_H$, then H is indistinguishable from S for every process, modulo possibly a last incomplete read operation.

Thus, S is a legal sequential history equivalent to a completion of H and preserves $\rightarrow_H$. $\Box_{Theorem\ 5}$

We say that a history H of a regular register commits a new/old inversion if H has a non atomic reading function. Notice that a history may have multiple reading functions, some atomic and some only regular. Theorem 4 and Theorem 5 imply that an atomic register is a regular register that prevents any new/old inversion.

Since linearizability is a local property, a set of 1WMR regular registers behave atomically if each prevents any new/old inversion.

4.3. Register transformations

In the following, we present several register transformations. Each transformation is an algorithm that builds a certain type of registers from a weaker type. The register constructed is called *high-level* whereas the ones we use are called *low-level* (or *base*). For example, we will show how to obtain a (high-level) regular register from (low-level) safe base registers, how to build a 1WMR register from 1W1R registers, or how to transform binary registers into a multi-valued one.

The transformations we present vary in their *complexity*, i.e., the number and size of the underlying base registers. For example, the number of base registers used by a transformation may be proportional to the number of readers. Also, a transformation may or not assume base registers of bounded capacity.

All transformations we present below are *wait-free*: every read or write operation of the high-level register terminates in a finite number of steps, some of these include reads and writes of the low-level register. Sometimes proving wait-freedom is trivial, e.g., when there is no loop or conditional statement. So we omit this proof.

In this and the next chapter, we proceed incrementally as follows.

1. We first present an algorithm that builds a 1WMR safe register out of 1W1R safe registers.

2. We then show how to build a binary 1WMR regular register out of a binary 1WMR safe register. Combining this algorithm with the one above, we obtain a transformation to a binary 1WMR regular register from binary 1W1R safe registers.

3. We present then transformations from binary 1WMR registers into multi-valued 1WMR registers that preserve the concurrency properties of the original one (be it safe, regular or atomic).

   All algorithms above are bounded. By combining them, we can for instance implement, in a bounded manner, a multi-valued 1WMR regular register using binary 1W1R safe registers.

   Later, in Chapter 5, we transform a 1W1R regular register into a MWMR atomic register by proceeding through three unbounded transformations as follows.

4. We show how to transform a 1W1R regular register into a 1W1R atomic register.

5. We present a transformation to a 1WMR atomic register from a 1W1R atomic register.

6. We show how to transform a 1WMR register into a MWMR register.

4.4. From one to multiple readers

We show here how to use single-reader (single-writer) safe registers to build a multi-reader (single-writer) safe register.

We consider a system of n processes, all being potential readers of the target high-level register. In the transformation, described in Figure 4.2, the constructed high-level register R is built from n 1W1R base registers, denoted *REG*$[1 : n]$, one associated with each reader process. Every reader p_i reads the base register *REG*$[i]$ it is associated with, while the single writer writes to every base register, one by one (in any order).

The transformation is bounded for it uses no information beyond the actual value stored: base registers can be of the same capacity as the multiple-reader register we construct.

```
operation $R.write(v)$:
    for_all $j$ in $\{1, \ldots, n\}$ do $REG[j] \leftarrow v$;
    return ()

operation $R.read()$ issued by $p_i$ :
    return ($REG[i]$)
```

Figure 4.2.: 1WMR safe from 1W1R safe

Theorem 6 *The algorithm of Figure 4.2 implements a 1WMR safe register using one safe 1W1R base register per reader.*

Proof Any read of the high-level register R (i.e., $R.read()$) that is not concurrent with any $R.write()$ operation returns the last value deposited in R, because of the safety of the underlying registers. The obtained register R is consequently safe while being 1WMR. $\Box_{Theorem\ 6}$

The very same transformation also works for regular registers.

Theorem 7 *The algorithm of Figure 4.2 implements a 1WMR regular register using one regular 1W1R base register per reader.*

Proof Since a regular register is safe, the previous theorem above implies that R is also safe. We thus need only show that a read operation $R.read()$ that is concurrent with one or more write operations returns a concurrently written value or the last written value.

Let p_i be any process that reads some value from R. When p_i reads the base regular register *REG*$[i]$ p_i returns (a) the value of a concurrent write on *REG*$[i]$ (if any) or (b) the last value written to *REG*$[i]$ before such concurrent write . In case (a), the value v obtained is from a $R.\mathit{write}(v)$ that is concurrent with the $R.\mathit{read}()$ of p_i. In case (b), v can either be (b.1) from a $R.\mathit{write}(v)$ that is concurrent with $R.\mathit{read}()$ of p_i , or (b.2) from the last value written by a $R.\mathit{write}()$ before the $R.\mathit{read}()$ of p_i. Thus, the constructed register R is regular. $\square_{Theorem\ 7}$

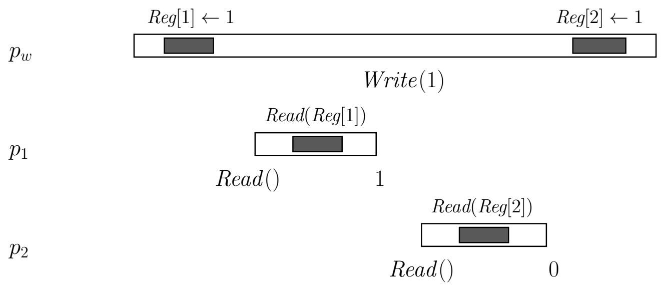


Figure 4.3.: New/old inversion

The algorithm of Figure 4.2, while indeed preserving safety and regularity, does not preserve atomicity. It does not implement a 1WMR atomic register even when every base register *REG*$[i]$ is a 1W1R atomic register. This is because of the possibility of new/old inversions (even if base registers prevent them). To illustrate this, consider the history conveyed in Figure 4.3. This history involves one writer p_w and two readers p_1 and p_2. Assume the high-level register R contains initially value 1 (*REG*$[1]$ and *REG*$[2]$ start with initial value 1). To write value 2 in R, the writer first performs *REG*$[1] \leftarrow 2$ and then *REG*$[2] \leftarrow 2$. Concurrently, p_1 reads *REG*$[1]$ and returns 2, and then p_2 reads *REG*$[2]$ and returns 1. This constitutes a new/old inversion: the read by p_1 returns the new value, and the subsequent read by p_2 returns the old value.

4.5. From a safe to a regular bit

We now construct a regular binary register using a single safe binary register. The algorithm heavily relies on the very fact that we can only store one of two values (0 or 1).

Remember that the difference between a safe and a regular register is only visible in the face of concurrency. That is, the value to be returned in the regular case has to be a value concurrently written or the last value written, while a safe register is allowed to return any value in the range of the register values (0 or 1 in the binary case). To understand the main idea behind the algorithm, consider a naive scheme where the regular register would be directly implemented using a safe base register. More precisely, assume every read (resp. write) on the high-level register is directly translated into a read (resp. write) on the base (safe) register. Assume furthermore the initial value is 0 and there is a write operation that writes the very same value 0. As the base register is only safe, a concurrent read operation could return value 1, which might have never been written. The high-level register would also return 1 and would violate regularity.

The problem is circumvented by preventing the writer from actually writing in the low-level register unless the writer intends to actually *change* the value of the high-level register. In this case of a change, the concurrent read could obtain any value in $\{0, 1\}$ (remember that only two values are possible), i.e., either the previously written or a concurrently written value, which ensures regularity.

In the algorithm, presented in Figure 4.4, besides a safe register *REG* shared between the reader and the writer, the writer maintains a local variable *prev_val* that contains the most recent value written in the base safe register *REG*. Before writing a value v in the high-level regular register, the writer checks if this value v is different from the previous value in *prev_val* and, only in that case, the writer writes v in *REG*.

```
operation $R.write(v)$:
    if ($prev\_val \neq v$) then $REG \leftarrow v$;
                                 $prev\_val \leftarrow v$;
    return ()

operation $R.read()$ issued by $p_i$ :
    return ($REG$)
```

Figure 4.4.: A binary regular from a binary safe register

Theorem 8 *The algorithm of Figure 4.4 implements a 1WMR binary regular register using an underlying 1WMR binary safe register.*

Proof Given that the underlying base register is safe, a read that is not concurrent with any write returns the last value written. Given also that the underlying base register always returns either 0 and 1 to any read, a read concurrent with one or more writes returns the value of the base register before these write operations or one of the values written by such a write operation. The high-level register is thus regular. $\square_{Theorem\ 8}$

If the safe base binary register is 1W1R, then the algorithm of Figure 4.4 also implements a 1W1R regular binary register.

It is important to see that the transformation does not implement an atomic register for it does not prevent new/old inversions. Assuming the writer writes value 1, a concurrent reader could return the new value 1 and then the old value 0. The transformation does clearly not work either for regular registers that store more than two values.

We present in the following three transformations from binary registers to emph$b$-valued registers, i.e., registers that can store a set of values of cardinality b.

The three transformations we present are all bounded, i.e., there is a bound on the number of base registers used, as well as on the amount of memory needed within each register. Also, the transformations preserve the concurrency semantics of the base registers: if the low-level register ensures concurrency property X (safe, regular or atomic), then so does the high-level (b-valued) register.

4.6. From safe bits to safe b-valued registers

The algorithm we present here uses a binary encoding scheme and hence several safe bits to implement a b-valued safe register R. The algorithm assumes that b, the capacity of R is a power of 2, i.e., $b = 2^B$ for some integer B. Any combination of B bits iis a value in the range of R.

The algorithm uses an array *REG*$[1 : B]$ of 1WMR safe bit registers to store the current value of R. Given $\mu_i =$ *REG*$[i]$, the binary representation of the current value of R is $\mu_1 \ldots \mu_B$. The algorithm is given in Figure 4.5.

Theorem 9 *Given B 1WMR safe bits, the algorithm of Figure 4.5 implements a 1WMR 2^B-valued safe register.*

```
operation $R.write(v)$:
    let $\mu_1 \ldots \mu_B$ be the binary representation of $v$;
    for_all $j$ in $\{1, \ldots, B\}$ do $REG[j] \leftarrow \mu_j$;
    return ()

operation $R.read()$ issued by $p_i$:
    for_all $j$ in $\{1, \ldots, B\}$ do $\mu_j \leftarrow REG[j]$;
    let $v$ be the value whose binary representation is $\mu_1 \ldots \mu_B$;
    return $(v)$
```

Figure 4.5.: A b-valued safe register from binary ones (binary encoding)

Proof Any read of R that does not overlap any write of R returns the binary representation of the last value written into R and is consequently safe to return. A read of R that overlaps a write of R can return any of b possible values whose binary encoding uses B bits. As every such combination represents one possible encoding of a value that R is supposed to contain, a read concurrent with a write indeed returns a value in the range of R. Consequently, R is a b-valued safe register. $\square_{Theorem\ 9}$

The space complexity of the algorithm is logarithmic with respect to the size b of the value range of the constructed register R.

4.7. From regular bits to regular b-valued registers

It is important to notice that the previous algorithm of Figure 4.5 does not implement a regular register even when the base registers are regular. Roughly speaking, this is because the write is not *continuous*. A read of R concurrent with a write changing for example the value of R from $0 \ldots 0$ to $1 \ldots 1$ can return any value, including one that was never written.

In order to ensure regularity, we use a different encoding scheme. Instead of a binary encoding as above, we turn to unary encoding: in short, whereas the former does not ensure the continuity of the writing, the latter does. Here, considering an array *REG*$[1 : b]$ of 1WMR regular bits, the value $v \in [1..b]$ is represented by 0s in registers 1 to $v - 1$ and then 1 in register at position v. The space complexity of the transformation algorithm is now b base bits, i.e., it is linear with respect to the size of the value range of the constructed register R instead of being logarithmic in the case of a safe register. This is the price we pay for regularity.

The algorithm that relies on unary encoding is given in Figure 4.6. To write v, the writer first sets *REG*$[v]$ to 1, and then *cleans* the array *REG*: this cleaning phase consists in setting to 0 all registers from *REG*$[v - 1]$ to *REG*$[1]$.

The reader traverses the array *REG*$[1 : b]$ starting from its first entry (*REG*$[1]$) and stops as soon as it finds an index j such that *REG*$[j] = 1$. The reader then returns j as the result of the read. Notice that a read searches for a value in the ascending order, while a write updates the array in the descending order, from $v - 1$ until 1. In other words, the write and the read are performed in the opposite directions.

The algorithm assumes that the register R has a valid initial value v_0: initially, *REG*$[j] = 0$ for $1 \le j < v_0$, *REG*$[v_0] = 1$, and *REG*$[j] = 0$ or 1 for $v_0 < j \le b$.

It is important to notice that, even when no write operation is in progress, several entries of the array may be set to 1. However, only the smallest entry of *REG* set to 1 actually encodes the most recently written value. The other entries are evidences of past values. Note also that the "last" base register *REG*$[b]$, once set to 1, does never change. A reader, once it witnessed 0 in all entries of *REG* up to $b - 1$, might thus by default consider *REG*$[b]$ to be 1.

We first argue for wait-freedom.

```
operation $R.write(v)$:
    $REG[v] \leftarrow 1$;
    for $j = v - 1$ down to $1$ do $REG[j] \leftarrow 0$;
    return ()

operation $R.read()$ issued by $p_i$:
    $j \leftarrow 1$;
    while $(REG[j] = 0)$ do $j \leftarrow j + 1$;
    return $(j)$
```

Figure 4.6.: A b-valued regular register from binary ones (unary-encoding)

Lemma 2 *The algorithm of Figure 4.6 is wait-free.*

Proof Every $R.write(v)$ operation terminates in a finite number of its own steps for its **for** loop only goes through v iterations.

Consider now a $R.read()$. Remember first that there initially is a valid value v_0 and hence initially a 1 in the register. Now observe also that whenever the writer changes $REG[x]$ from 1 to 0, the writer has already set to 1 another entry $REG[y]$ such hat $x < y \leq b$. Hence, if a reader reads $REG[x]$ and returns the new value 0, then a higher entry of the array has been set to 1. As the index of the **while** loop of the read starts at 1 and is incremented each time the loop body is performed, the loop eventually terminates in a finite number of steps. $\Box_{Lemma\ 2}$

The previous lemma relies heavily on the fact that the high-level register R can contain up to b distinct values. If the range of R is unbounded, a $R.read()$ operation might never terminate if the writer continuously updates R with ever-increasing values. More precisely, suppose that the range of R is unbounded and consider the following scenario. Let $R.write(x)$ be the last write operation terminated before a $R.read()$ starts. Let the read operation proceed until it is about to read $REG[x]$ and then schedule a concurrent $R.write(y)$, $y > x$) to set $REG[x]$ from 1 to 0. Then we schedule the read of $REG[x]$ by the reader. As the register is unbounded, this scenario can repeat indefinitely, forcing the reader to take infinitely many reads of *REG*.

Theorem 10 *Given b 1WMR regular bits, the algorithm described in Figure 4.6 implements a 1WMR b-valued regular register.*

Proof Consider first a read operation that is not concurrent with any write, and let v be the last written value. By the write algorithm, when the corresponding $R.write(v)$ terminates, the first entry of the array that equals 1 is $REG[v]$ (i.e., $REG[x] = 0$ for $1 \leq x \leq v - 1$). Because a read traverses the array starting from $REG[1]$, then $REG[2]$, etc., it necessarily reads until $REG[v]$ and returns the value v.

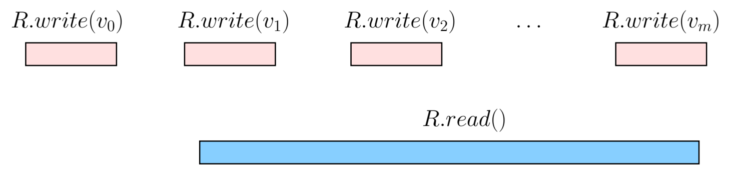


Figure 4.7.: A read with concurrent writes

Let us now consider a read operation $R.read()$ that is concurrent with one or more write operations $R.write(v_1)$, ..., $R.write(v_m)$ (as depicted in Figure 4.7). Let v_0 be the value written by the last write operation that terminated before the operation $R.read()$ starts. For simplicity we assume that each execution begins with a write operation that sets the value of R to an initial value. As a read operation always terminates (Lemma 2), the number of writes concurrent with the $R.read()$ operation is finite.

By the algorithm, the read operation finds 0 in *REG*$[1]$ up to *REG*$[v-1]$, 1 in *REG*$[v]$, and then returns v. We are going to show by induction that each of these base-object reads returns a value previously or concurrently written by a write operation in $R.write(v_0)$, $R.write(v_1)$, ..., $R.write(v_m)$.

Since $R.write(v_0)$ sets *REG*$[v_0]$ to 1 and *REG*$[v_0 - 1]$ down to *REG*$[1]$ to 0, the first base-object read performed by the $R.read()$ operation returns the value written by $R.write(v_0)$ or a concurrent write. Now suppose that the read on *REG*$[j]$, for some $j = 1, \dots, v-1$, returned 0 written by the latest preceding or a concurrent write operation $R.write(v_k)$ $(k = 1, \dots, m)$. Notice that $v_k > j$: otherwise, $R.write(v_k)$ would not touch *REG*$[j]$. By the algorithm, $R.write(v_k)$ has previously set *REG*$[v_k]$ to 1 and *REG*$[v_k - 1]$ down to *REG*$[j+1]$ to 0. Thus, since the base registers are regular, the subsequent read of *REG*$[j+1]$ performed within the $R.read()$ operation can only return the value written by $R.write(v_k)$ or a subsequent write operation that is concurrent with $R.read()$.

By induction, we derive that the read of *REG*$[v]$ performed within $R.read()$ returns a value written by the latest preceding or a concurrent write. $\square_{Theorem\ 10}$

4.8. From atomic bits to atomic b-valued registers

In Chapter 6, we give a direct construction of an atomic bit from three regular ones. However, if we use this construction to replace regular bits with atomic ones in the algorithm in Figure 4.6 we do not get an atomic b-valued register. Interestingly, a relatively simple modification of its read algorithm makes that possible by preventing the new/old inversion phenomenon.

The idea is to equip the $R.read()$ algorithm in Figure 4.6 with a "counter-inversion" mechanism. Instead of returning position j where the first 1 was located in *REG*, the read operation traverses the array again in the opposite direction (from j to 1) and returns the smallest entry containing value 1. The resulting algorithm is presented in Figure 4.8.

```
    operation $R.write(v)$:
        $REG[v] \leftarrow 1$;
        for $j$ from $v-1$ step $-1$ until $1$ do $REG[j] \leftarrow 0$ ;
        return ()

    operation $R.read()$ issued by $p_i$:
        $j\_up \leftarrow 1$;
(1)     while $(REG[j\_up] = 0)$ do $j\_up \leftarrow j\_up + 1$;
(2)     $j \leftarrow j\_up$;
(3)     for $j\_down$ from $j\_up - 1$ step $-1$ until $1$ do
(4)                         if $(REG[j\_down] = 1)$ then $j \leftarrow j\_down$
        return $(j)$
```

Figure 4.8.: Atomic register: from bits to b-valued register

Theorem 11 *The algorithm in Figure 4.8 implements a 1WMR atomic b-valued register using b 1WMR atomic bits.*

Proof For every history of the algorithm, we define the reading function π as follows. Let r be a read operation that returned v. Then $\pi(r)$ is the latest write operation that updated *REG*$[v]$ before the last read of *REG*$[v]$ performed by r, or the initializing write operation w_0 if no such operation exists. Since

r returns the index of *REG* containing 1, $\pi(r)$ writes 1 to *REG*$[v]$. Note that π is well-defined, as it can be derived from the atomic reading function of the elements of *REG*.

We now show that π is indeed an atomic reading function, i.e., it satisfies properties $A1$, $A2$ and $A3$ in Section 4.2. By the definition, $\pi(r)$ is a preceding or concurrent write operation, therefore $A1$ is satisfied.

To see that $A2$ is also satisfied, suppose, by contradiction, that $\pi(r) \to w(v') \to r(v)$ for some write $w(v')$. By the algorithm, $w(v')$ sets *REG*$[v]$ to 1 and then writes 0 to all *REG*$[v-1]$ down to *REG*$[1]$. Thus, $v' < v$, otherwise $w(v')$ would also write to *REG*$[v]$ and $\pi(r)$ would not be the latest write updating *REG*$[v]$ before r reads *REG*$[v]$. Since r reached *REG*$[v]$, there exists a write $w(v'')$ that set *REG*$[v']$ to 0 after $w(v')$ set it to 1 but before r read it. By the algorithm, before setting *REG*$[v']$ to 0 this write has set a *REG*$[v'']$ to 1 and, by the assumption, $v'' < v$. Assuming that $w(v'')$ is the latest such write, before reacing *REG*$[v]$, r must have found *REG*$[v''] = 1$—a contradiction.

To show that π satisfies $A3$, let us consider two read operations $r1$ and $r2$, $r1 \to r2$, and suppose, by contradiction, that $\pi(r2) \to \pi(r1)$.

Let $r1$ return v and $r2$ return v'. Since $\pi(r1) \neq \pi(r1)$, the definition of π implies that $v \neq v'$. Thus, we should only consider the following cases:

(1) $v' > v$.

In this case, $r2$ must have found 0 in *REG*$[v]$ before finding 1 in *REG*$[v']$ and returning $v' > v$. Thus, a write $w(v'')$ such that $v < v'' < v'$ and $\pi(r2) \to w(v'') \to (r1)$, has set *REG*$[v]$ to 0 after $\pi(v)$ set *REG*$[v]$ to 1 but before $r2$ read it. Assume, without loss of generality, that v'' is the smallest such value. Since $w(v'')$ has set *REG*$[v'']$ to 1 before writing 0 to *REG*$[v]$, $r2$ must have returned $v'' < v'$—a contradiction.

(2) $v' < v$.

In this case, $r1$ reads 1 in *REG*$[v]$, $v > v'$, and then reads 0 in all *REG*$[v-1]$ down to *REG*$[1]$, including *REG*$[v']$. Since $\pi(r2)$ has previously set *REG*$[v']$ to 1, another write operation must have set *REG*$[v']$ to 0 after $\pi(r2)$ set it to 1 but before $r1$ read it. Thus, when $r2$ subsequently reads 1 in *REG*$[v']$, $\pi(r2)$ is not the last preceding write operation to write to *REG*$[v']$—a contradiction with the definition of π.

Hence, π is an atomic reading function and, by Theorem 5, the algorithm indeed implements a 1WMR atomic register. $\square_{Theorem\ 11}$

4.9. Bibliographic notes

The notions of safe, regular and atomic registers have been introduced by Lamport [49].

Theorem 5, and the algorithms described in Figure 4.2, Figure 4.4, Figure 4.5 and Figure 4.6 are due to Lamport [49]. The algorithm described in Figure 4.8 is due to Vidyasankar [69].

The wait-free construction of stronger registers from weaker registers has always been an active research area. The interested reader can consult the following (non-exhaustive!) list where numerous algorithms are presented and analyzed [11, 14, 17, 18, 33, 42, 51, 65, 70, 71, 72].

4.10. Exercises

1. Multi-valued regular registers.

Consider the implementation of an M-valued one-writer N-reader ($1WNR$) regular register (Figure 4.6).

a) In the code of *write(v)*, is it possible to change the order of operations: first write 0 to *REG*$[v-1], \ldots,$ *REG*$[1]$ and then write 1 to *REG*$[v]$?

b) What if the writer writes 0 to *REG*$[1], \ldots,$ *REG*$[v-1]$ in the ascending order? Justify your answers (e.g., by presenting an execution that violates the properties of a regular register).

2. Multi-valued atomic registers.

   a) In the algorithm in Figure 4.6, if we replace the regular binary registers with *atomic* ones, would we get an implementation of an atomic multi-valued register?

   b) If we replace the regular binary registers with *atomic* ones, would we get an implementation of an atomic multi-valued register?

5. Unbounded register transformations

In this chapter we consider a simplistic case when *unbounded* base objects, i.e., registers of unbounded capacity, can be used. This assumption allows us to use the *sequence numbers*: each written value is associated with a sequence number that intuitively captures the number of write operations performed up to now. A typical base register consists therefore of two fields: a data field that stores the value of the register and a control field that stores the sequence number associated with it.

Of course, assuming base objects of unbounded capacity is not very realistic. In the coming Chapters 6 and 7 we discuss algorithms that implement *bounded* (i.e., storing values from a bounded range) atomic registers using bounded safe registers.

5.1. 1W1R registers: From unbounded regular to atomic

We show in the following how to implement an 1W1R atomic register using a 1W1R regular register. The use of sequence numbers make such a construction easy and helps in particular prevent the new/old inversion phenomenon. Preventing this, while preserving regularity, means, by Theorem 5, that the constructed register is atomic.

The algorithm is described in Figure 5.1. Exactly one base regular register *REG* is used in the implementation of the high-level register R. The local variable sn at the writer is used to hold sequence numbers. It is incremented for every new write in R. The scope of the local variable aux used by the reader spans a read operation; it is made up of two fields: a sequence number ($aux.sn$) and a value ($aux.val$).

Each time it writes a value v in the high-level register, R, the writer writes the pair $[sn, v]$ in the base regular register *REG*. The reader manages two local variables: $last\_sn$ stores the greatest sequence number it has even read in *REG*, and $last\_val$ stores the corresponding value. When it wants to read R, the reader first reads *REG*, and then compares $last\_sn$ with the sequence number it has just read in *REG*. The value with the highest sequence number is the one returned by the reader and this prevents new/old inversions.

```
operation $R.write(v)$:
    $sn \leftarrow sn + 1$;
    $REG \leftarrow [sn, v]$;
    return ()

operation $R.read()$:
    $aux \leftarrow REG$;
    if ($aux.sn > last\_sn$) then $last\_sn \leftarrow aux.sn$;
                                  $last\_val \leftarrow aux.val$;
    return ($last\_val$)
```

Figure 5.1.: From regular to atomic: unbounded construction

Theorem 12 *Given an unbounded 1W1R regular register, the algorithm described in Figure 5.1 constructs a 1W1R atomic register.*

Proof The proof is similar to the proof of Theorem 5. We associate with each read operation r of the high-level register R, the sequence number (denoted $sn(r)$) of the value returned by r: this is possible as the base register is regular and consequently a read always returns a value that has been written with its sequence number, that value being the last written value or a value concurrently written (if any). Considering an arbitrary history H of register R, we show that H is atomic by building an equivalent sequential history S that is legal and respects the partial order on the operations defined by $\rightarrow_H$.

S is built from the sequence numbers associated with the operations. First, we order all the write operations according to their sequence numbers. Then, we order each read operation just after the write operation that has the same sequence number. If two reads operations have the same sequence number, we order first the one whose invocation event is first. (Remember that we consider a 1W1R register.)

The history S is trivially sequential as all the operations are placed one after the other. Moreover, S is equivalent to H as it is made up of the same operations. S is trivially legal as each read follows the corresponding write operation. We now show that S respects $\rightarrow_H$.

- For any two write operations $w1$ and $w2$ we have either $w1 \rightarrow_H w2$ or $w2 \rightarrow_H w1$. This is because there is a single writer and it is sequential: as the variable sn is increased by 1 between two consecutive write operations, no two write operations have the same sequence number, and these numbers agree on the occurrence order of the write operations. As the total order on the write operations in S is determined by their sequence numbers, it consequently follows their total order in H.

- Let $op1$ be a write or a read operation, and $op2$ be a read operation such that $op1 \rightarrow_H op2$. It follows from the algorithm that $sn(op1) \leq sn(op2)$ (where $sn(op)$ is the sequence number of the operation op). The ordering rule guarantees that $op1$ is ordered before $op2$ in S.

- Let $op1$ be a read operation, and $op2$ a write operation. Similarly to the previous item, we then have $sn(op1) < sn(op2)$, and consequently $op1$ is ordered before $op2$ in S (which concludes the proof).

$\Box_{Theorem\ 12}$

One might think of a naïve extension of the previous algorithm to construct a 1WMR atomic register from base 1W1R regular registers. Indeed, we could, at first glance, consider an algorithm associating one 1W1R regular register per reader, and have the writer writes in all of them, each reader reading its dedicated register. Unfortunately, a fast reader might see a new concurrently written value, whereas a reader that comes later sees the old value. This is because the second reader does not know about the sequence number and the value returned by the first reader. The latter stores them locally. In fact, this can happen even if the base 1W1R registers are atomic. The construction of a 1WMR atomic register from base 1W1R atomic registers is addressed in the next section.

5.2. Atomic registers: from unbounded 1W1R to 1WMR

In Section 4.4, we presented an algorithm that builds a 1WMR safe/regular register from similar 1W1R base registers. We also pointed out that the corresponding construction does not build a 1WMR atomic register even when the base registers are 1W1R atomic (see the counter-example presented in Figure 4.3).

This section describes such an algorithm: assuming 1W1R atomic registers, it shows how to go from single reader registers to a multi-reader register. This algorithm uses sequence numbers, and requires unbounded base registers.

Overview. As there are now several possible readers, actually n, we make use of several (n) base 1W1R atomic registers: one per reader. The writer writes in all of them. It writes the value as well as a sequence number. The algorithm is depicted in Figure 5.2.

We prevent new/old inversions (Figure 4.3) by having the readers "help" each other. The helping is achieved using an array *HELP*$[1:n, 1:n]$ of 1W1R atomic registers. Each register contains a pair (sequence number, value) created and written by the writer in the base registers. More specifically, *HELP*$[i, j]$ is a 1W1R atomic register written only by p_i and read only by p_j. It is used as follows to ensure the atomicity of the high-level 1WMR register R that is constructed by the algorithm.

- *Help the others.* Just before returning the value v it has determined (we discuss how this is achieved in the second bullet below), reader p_i helps every other process (reader) p_j by indicating to p_j the last value p_i has read (namely v) and its sequence number sn. This is achieved by having p_i update *HELP*$[i, j]$ with the pair $[sn, v]$. This, in turn, prevents p_j from returning in the future a value older than v, i.e., a value whose sequence number would be smaller than sn.

- *Helped by the others.* To determine the value returned by a read operation, a reader p_i first computes the greatest sequence number that it has ever seen in a base register. This computation involves all 1W1R atomic registers that p_i can read, i.e., *REG*$[i]$ and *HELP*$[j, i]$ for any j. p_i. Reader p_i then returns the value that has the greatest sequence number p_i has computed.

The corresponding algorithm is described in Figure 5.2. Variable aux is a local array used by a reader; its jth entry is used to contain the (sequence number, value) pair that p_j has written in *HELP*$[j, i]$ in order to help p_i; $aux[j].sn$ and $aux[j].val$ denote the corresponding sequence number and the associated value, respectively. Similarly, *reg* is a local variable used by a reader p_i to contain the last (sequence number, value) pair that p_i has read from *REG*$[i]$ (*reg*$.sn$ and *reg*$.val$ denote the corresponding fields).

Register *HELP*$[i, i]$ is used only by p_i, which can consequently keep its value in a local variable. This means that the 1W1R atomic register *HELP*$[i, i]$ can be used to contain the 1W1R atomic register *REG*$[i]$. It follows that the protocol requires exactly n^2 base 1W1R atomic registers.

```
operation $R.write(v)$:
    $sn \leftarrow sn + 1$;
    for_all $j$ in $\{1, \dots, n\}$ do $REG[i] \leftarrow [sn, v]$;
    return ()

operation $R.read()$ issued by $p_i$:
    $reg \leftarrow REG[i]$;
    for_all $j$ in $\{1, \dots, n\}$ do $aux[j] \leftarrow HELP[j, i]$;
    let $sn\_max$ be $\max(reg.sn, aux[1].sn, \dots, aux[n].sn)$;
    let $val$ be $reg.val$ or $aux[k].val$ such that the associated seq number is $sn\_max$;
    for_all $j$ in $\{1, \dots, n\}$ do $HELP[i, j] \leftarrow [sn\_max, val]$;
    return ($val$)
```

Figure 5.2.: Atomic register: from one reader to multiple readers (unbounded construction)

Theorem 13 *Given n^2 unbounded 1W1R atomic registers, the algorithm described in Figure 5.2 implements a 1WMR atomic register, where n is the number of readers.*

Proof As for Theorem 5, the proof consists in showing that the sequence numbers determine a linearization of any history H.

Considering an history H of the constructed register R, we first build an equivalent sequential history S by ordering all the write operations according to their sequence numbers, and then inserting the read operations as in the proof of Theorem 5. This history is trivially legal as each read operation is ordered just after the write operation that wrote the value that is read. A reasoning similar to the one used in Theorem 5, but based on the sequence numbers provided by the arrays *REG*$[1:n]$ and *HELP*$[1:n, 1:n]$, shows that S respects $\rightarrow_H$. $\Box_{Theorem\ 18}$

5.3. Atomic registers: from unbounded 1WMR to MWMR

In this section, we show how to use sequence numbers to build a MWMR atomic register from n 1WMR atomic registers (where n is the number of writers). The algorithm is simpler than the previous one. An array *REG*$[1:n]$ of n 1WMR atomic registers is used in such a way that p_i is the only process that can write in *REG*$[i]$, while any process can read it. Each register *REG*$[i]$ stores a (sequence number, value) pair. Variables $X.sn$ and $X.val$ are again used to denote the sequence number field and the value field of the register X, respectively. Each *REG*$[i]$ is initialized to the same pair, namely, $[0, v_0]$ where v_0 is the initial value of R.

The problem we solve here consists in allowing the writers to totally order their write operations. To that end, a write operation first computes the highest sequence number that has been used, and defines the next value as the sequence number of its write. Unfortunately, this does not prevent two distinct concurrent write operations from associating the same sequence number with their respective values. A simple way to cope with this problem consists in associating a *timestamp* with each value, where a timestamp is a pair of a sequence number and the identity of the process that issues the corresponding write operation.

The timestamping mechanism can be used to define a total order on all the timestamps as follows. Let $ts1 = [sn1, i]$ and $ts2 = [sn2, j]$ be any two timestamps. We have:

$$ts1 < ts2 \overset{\text{def}}{=} \big((sn1 < sn2) \vee (sn1 = sn2 \wedge i < j)\big).$$

The corresponding construction is described in Figure 5.3. The meaning of the additional local variables that are used is, we believe, clear from the context.

```
operation $R.write(v)$ issued by $p_i$:
    for_all $j$ in $\{1, \ldots, n\}$ do $reg[j] \leftarrow REG[j]$;
    let $sn\_max$ be $\max(reg[1].sn, \ldots, reg[n].sn) + 1$;
    $REG[i] \leftarrow [sn\_max, v]$;
    return ()

operation $R.read()$ issued by $p_i$:
    for_all $j$ in $\{1, \ldots, n\}$ do $reg[j] \leftarrow REG[j]$;
    let $k$ be the process identity such that $[sn, k]$ is the greatest timestamp
        among the $n$ timestamps $[reg[1].sn, 1]$, ... and $[reg[n].sn, n]$;
    return ($reg[k].val$)
```

Figure 5.3.: Atomic register: from one writer to multiple writers (unbounded construction)

Theorem 14 *Given n unbounded 1WMR atomic registers, the algorithm described in Figure 5.3 implements a MWMR atomic register.*

Proof Again, we show that the timestamps define a linearization of any history H.

Considering an history H of the constructed register R, we first build an equivalent sequential history S by ordering all the write operations according to their timestamps, then inserting the read operations as in Theorem 5. This history is trivially legal as each read operation is ordered just after the write operation that wrote the read value. Finally, a reasoning similar to the one used in Theorem 5 but based on timestamps shows that S respects $\rightarrow_H$. $\Box_{Theorem\ 14}$

5.4. Concluding remark

The algorithms presented in this chapter assume that the sequence numbers may grow without bound, hence the assumption of unbounded base registers. This appears like wasting resources in the case when the values written to the implemented register are taken from a bounded range.

On approach to bound the capacity of base registers is based on *timestamp systems*. These techniques, originally proposed by Dolev and Shavit [23] and Dwork and Waarts [24], emulate shared sequence numbers taken from a fixes range, bounded by a function of the number of processes. A prominent atomic register construction based on bounded timestamps was proposed by Li, Tromp, and Vitanyi [51].

In Chapters 6 and 7, we discuss an alternative, less generic but simpler, solution based on elementary binary *signalling* between the writer and the reader in the one-reader case 6), and, additionally, between the readers in the multiple-readers case (Chapter 7). Also, in Chapter 8, we discuss how to implement the bounded *atomic snapshot* abstraction *directly*, using registers of bounded capacity.

5.5. Bibliographic notes

The notions of safe, regular and atomic registers have been introduced by Lamport [49].

Theorem 5, and the algorithms described in Figure 4.2, Figure 4.4, Figure 4.5 and Figure 4.6 are due to Lamport [49]. The algorithm described in Figure 4.8 is due to Vidyasankar [69]. The algorithms described in Figure 5.2 and 5.3 are due to Vityani and Awerbuch [73].

The wait-free construction of stronger registers from weaker registers has always been an active research area. The interested reader can consult the following (non-exhaustive!) list where numerous algorithms are presented and analyzed [11, 14, 17, 18, 33, 42, 51, 65, 70, 71, 72].

5.6. Exercises

1. Give an example of a history of a read-write atomic register that allows for a regular but not atomic reading function.

2. Prove that the implementation of a one-writer one-reader ($1W1R$) atomic register is correct (Transformation IV in the slides).

   Hint: argue that to prove that the implementation is indeed linearizable, it is enough to show that if $read_1$ precedes $read_2$, then $read_2$ cannot return the value written before the value returned by $read_1$. Check the claim and the rest is trivial.

3. Consider the implementation of a one-writer N-reader ($1WNR$) atomic register (Transformation V in the slides).

   The code of *read()* involves writing the value just read back to $RR[\,][\,]$. Is it possible to devise an implementation in which the reader *does not* write?

4. Give a *multi-writer* multi-reader ($NWNR$) atomic register implementation from $1W1R$ atomic registers and sketch a proof of its correctness.

6. Optimal atomic bit construction

6.1. Introduction

In the previous chapter, we introduced the notions of safe, regular and atomic (linearizable) read/write objects (also called registers). In the case of 1W1R (one writer one reader) register, assuming that there is no concurrency between the reader and the writer, the notions of safety, regularity and atomicity are equivalent. This is no longer true in the presence of concurrency. Several bounded constructions have been described for concurrent executions. Each construction implements a stronger register from a collection of weaker base registers. We have seen the following constructions:

- From a safe bit to a regular bit. This construction improves on the quality of the base object with respect to concurrency. Contrarily to the base safe bit, a read operation on the constructed regular bit never returns an arbitrary value in presence of concurrent write operations.

- From a bounded number of safe (resp., regular or atomic) bits to a safe (resp., regular or atomic) b-valued register. These constructions improve on the quality of each base object as measured by the number of values it can store. They show that "small" base objects can be composed to provide "bigger" objects that have the same behavior in the presence of concurrency.

To get a global picture, we miss one bounded construction that improves on the quality in the presence of concurrency, namely, a construction of an atomic bit from regular bits. This construction is fundamental, as an atomic bit is the simplest nontrivial object that can be defined in terms of *sequential* executions. Even if an execution on an atomic bit contains concurrent accesses, the execution still appears as its sequential *linearization*.

In this chapter, we first show that to construct a 1W1R atomic bit, we need at least three safe bits, two written by the writer and one written by the reader. Then we present an optimal three-bit construction of an atomic bit.

6.2. Lower bound

In Section 5.1, we presented the construction of a 1W1R atomic register from an *unbounded* regular register. The base regular register had to be unbounded because the construction was using sequence numbers, and the value of the base register was a pair made up of the data value of the register and the corresponding sequence number. The use of sequence numbers makes sure that new-old inversions of read operations never happen.

A fundamental question is the following: Can we build a 1W1R atomic register from a finite number of regular registers that can store only finitely many values, and can be written only by the writer (of the atomic register)?

This section first shows that such a construction is impossible, i.e., the reader must also be able to write. In other words, such a construction must involve two-way communication between the reader and the writer. Moreover, even if we only want to implement one atomic bit, the writer must be able to write in *two* regular base bits.

6.2.1. Digests and sequences of writes

Let A be any finite sequence of values in a given set. A *digest* of A is a shorter sequence B that "mimics" A: A and B have the same first and last elements; an element appears at most once in B; and two consecutive elements of B are also consecutive in A. B is called a *digest* of A.

As an example let $A = v_1, v_2, v_1, v_3, v_4, v_2, v_4, v_5$. The sequence $B = v_1, v_3, v_4, v_5$ is a digest of A. (there can be multiple digests of a sequence).

Every finite sequence has a digest:

Lemma 3 *Let $A = a_1, a_2, \ldots, a_n$ be a finite sequence of values. For any such sequence there exists a sequence $B = b_1, \ldots, b_m$ of values such that:*

- $b_1 = a_1 \ \wedge \ b_m = a_n$,
- $(b_i = b_j) \Rightarrow (i = j)$,
- $\forall j : \ 1 \leq j < m : \exists i : \ 1 \leq i < n : b_j = a_i \ \wedge \ b_{j+1} = a_{i+1}$.

Proof The proof is a trivial induction on n. If $n = 1$, we have $B = a_1$. If $n > 1$, let $B = b_1, \ldots, b_m$ be a digest of $A = a_1, a_2, \ldots, a_n$. A digest of $a_1, a_2, \ldots, a_n, a_{n+1}$ can be constructed as follows:
- If $\forall j \in \{1, \ldots, m\} : b_j \neq a_{n+1}$, then $B = b_1, \ldots, b_m, a_{n+1}$ is a digest of $a_1, a_2, \ldots, a_n$.
- If $\exists j \in \{1, \ldots, m\} : b_j = a_{n+1}$, there is a single j such that $b_j = a_{n+1}$ (this is because any value appears at most once in $B = b_1, \ldots, b_m$). It is easy to check that $B = b_1, \ldots, b_j$ is a digest of $a_1, \ldots, a_n, a_{n+1}$. $\square_{Lemma\ 3}$

Consider now an implementation of a bounded atomic 1W1R register R from a collection of base *bounded* 1W1R regular registers. Clearly, any execution of a write operation w that changes the value of the implemented register must consist of a sequence of writes on base registers. Such a sequence of writes triggers a sequence of state changes of the base registers, from the state before w to the state after w.

Assuming that R is initialized to 0, let us consider an execution E where the writer indefinitely alternates $R.write(1)$ and $R.write(0)$. Let w_i, $i \geq 1$, denotes the i-th $R.write(v)$ operation. This means that $v = 1$ when i is odd and $v = 0$ when i is even. Each prefix of E, denoted by E', unambiguously determines the resulting *state* of each base object X, i.e., the value that the reader would obtain if it read X right after E', assuming no concurrent writes. Indeed, since the resulting execution is sequential, there exists exactly one reading function and we can reason about the state of each object at any point in the execution.

Each write operation $w_{2i+1} = R.write(1)$, $i = 0, 1, \ldots$, contains a sequence of writes on the base registers. Let $\omega_1, \ldots, \omega_x$ be the sequence of base writes generated by w_{2i+1}. Let A_i be the corresponding sequence of base-registers states defined as follows: its first element a_0 is the state of the base registers before ω_1, its second element a_2 is the state of the base registers just after ω_1 and before ω_2, etc.; its last element a_x is the state of the base registers after ω_x.

Let B_i be a digest derived from A_i (by Lemma 3 such a digest sequence exists).

Lemma 4 *There exists a digest $B = b_0, \ldots, b_y$ ($y \geq 1$) that appears infinitely often in $B_1, B_2, \ldots$.*

Proof First we observe that every digest B_i ($i = 1, 2, \ldots$) must consists of at least two elements. Indeed if B_i is a singleton b_0, then the read operation on R applied just before w_i and the read operation on R applied just after w_i observe the same state of base registers b_0. Therefore, the reader cannot decide when exactly the read operation was applied and must return the same value—a contradiction with the assumption that w_i changes the value of R.

Since the base registers are bounded, there are finitely many different states of the base registers that can be written by the writer. Since a digest is a sequence of states of the registers written by the writer in which every state appears at most once, we conclude that there can only be finitely many digests. Thus, in the infinite sequence of digests, $B_1, B_2, \ldots$, some digest B (of two or more elements) must appear infinitely often. $\Box_{Lemma\ 4}$

Note that there is no constraint on the number of *internal* states of the writer. Since there may be no bound on the number of steps taken within a write operation, all the sequences A_i can be different, and the writer may never perform the same sequence of base-register operations twice. But the evolution of the base-register states in the course of A_i can be reduced to its digest B_i.

6.2.2. Impossibility result and lower bound

Theorem 15 *It is not possible to build a 1W1R atomic bit from a finite number of regular registers that can take a finite number of values and are written only by the writer.*

Proof By contradiction, assume that it is possible to build a 1W1R atomic bit R from a finite set S of regular registers, each with a finite value domain, in which the reader does not update base registers.

An operation $r = R.read()$ performed by the reader is implemented as a sequence of read operations on base registers. Without loss of generality, assume that r reads *all* base registers. Consider again the execution E in which the writer performs write operations $w_1, w_2, \ldots$, alternating $R.write(1)$ and $R.write(0)$.

Since the reader does not update base registers, we can insert the complete execution of r between every two steps in E without affecting the steps of the writer. Since the base registers are regular, the value read in a base register X by the reader performing r after a prefix of E is unambiguously defined by the latest value written to X before the beginning of r. Let $\lambda(r)$ denote the state of all base registers observed by r.

By Lemma 4, there exists a digest $B = b_0, \ldots, b_y$ ($y \geq 1$) that appears infinitely often in $B_1, B_2, \ldots$, where B_i is a digest of w_{2i+1}. Since each state in $\{b_0, \ldots, b_y\}$ appears in E infinitely often, we can construct an execution E' by inserting in E a sequence of read operations $r_0, \ldots, r_y$ such that for each $j = 0, \ldots, y$, $\lambda(r_j) = b_{y-j}$. In other words, in E', the reader observes the states of base registers evolving downwards from b_y to b_0.

By induction, we show that in E', each r_j ($j = 0, \ldots, y$) must return 1. Initially, since $\lambda(r_0) = b_y$ and b_y is the state of the base registers right after some $R.write(1)$ is complete, r_0 must return 1. Inductively, suppose that r_j (for some j, $0 \leq j \leq y-1$) returns 1 in E'.

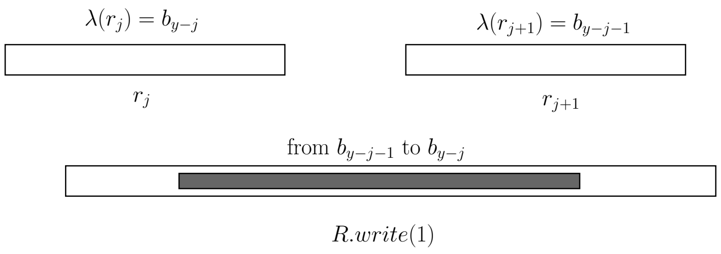


Figure 6.1.: Two read operations r_j and $r_j + 1$ concurrent with $R.write(1)$

Consider read operations r_j and r_{j+1} ($j = 0, \ldots, y-1$). Recall that $\lambda(r_j) = b_{y-j}$ and $\lambda(r_{j+1}) = b_{y-j-1}$. Since digest B appears in $B_1, B_2, \ldots$ infinitely often, E' contains infinitely many base-register writes by which the writer changes the state of base registers from b_{y-j-1} to b_{y-j}. Let X be the base register changed by these writes.

Since X is regular, we can construct an execution E'' which is indistinguishable to the reader from E', where r_j are concurrent with a base-register write performed within $R.write(1)$ in which the writer changes the state of the base registers from b_{y-j-1} to $b_y - j$ (Figure 6.1).

By the induction hypothesis, r_j returns 1 in E' and, thus, in E''. Since the implemented register R is atomic and r_j returns the concurrently written value 1 in E'', r_{j+1} must also return 1 in E''. But the reader cannot distinguish E' and E'' and, thus, r_{j+1} returns 1 also in E'.

Inductively, r_y must return 1 in E'. But $\lambda(r_y) = b_0$, where b_0 is the state of base registers right after some $R.write(0)$ is complete. Thus, r_y must return 0—a contradiction. $\Box_{Theorem\ 15}$

Therefore, to implement a 1W1R atomic register from bounded regular registers, we must establish two-way communication between the writer and the reader. Intuitively, the reader must inform the writer that it is aware of the latest written value, which requires at least one base bit that can be written by the reader and read by the writer. But the writer must be able to react to the information read from this bit. In other words:

Theorem 16 *In any implementation a 1W1R atomic bit from regular bits, the writer must be able to write to at least 2 regular bits.*

Proof Suppose, by contradiction, that there exists an implementation of a 1W1R atomic bit R in which the writer can write to exactly one base bit X.

Note that every write operation on R that changes the value of X and does not overlap with any read operation must change the state of X. Without loss of generality assume that the first write operation $w_1 = R.write(1)$ performed by the writer in the absence of the reader changes the value of X from 0 to 1 (the corresponding digest is $0, 1$).

Consider an extension of this execution in which the reader performs $r_1 = R.read()$ right after the end of w_1. Clearly, r_1 must return 1. Now add $w_2 = R.write(0)$ right after the end of r_1. Since the state of X at the beginning of w_2 is 1, the only digest generated by w_2 is $1, 0$.

Now add $r_2 = R.read()$ right after the end of w_2, and let E be the resulting execution. Now r_2 must return 0 in E. But since X is regular, E is indistinguishable to the reader from an execution in which r_1 and r_2 take place within the interval of w_1 and thus both must return 1—a contradiction. $\Box_{Theorem\ 16}$

As we have seen in the previous chapter, there is a trivial bounded algorithm that constructs a regular bit from a safe bit. This algorithm only requires one additional local variable at the writer. The combination of this algorithm with Theorem 16 implies:

Corollary 1 *The construction of a 1W1R atomic bit from safe bits requires at least 3 1W1R safe bits, two written by the writer and one written by the reader.*

As the construction presented in the next section uses exactly 3 1W1R regular bits to build an atomic bit, it is optimal in the number of base safe bits.

6.3. From three safe bits to an atomic bit

Now we present an optimal construction of a high level 1W1R atomic bit R from three base 1W1R safe bits. The high level bit R is assumed to be initialized to 0. It is also assumed that each $R.write(v)$ operation invoked by the writer changes the value of R. This is done without loss of generality, as the writer of R can locally keep a copy v' of the last written value, and apply the next $R.write(v)$ operation only when it modifies the current value of R.

The construction of R is presented in an incremental way.

6.3.1. Base architecture of the construction

The three base registers are initialized to 0. Then, as we will see, the read and write algorithms defining the construction, are such that, any write applied to a base register X changes its value. So, its successive values are 0, then 1, then 0, etc. Consequently, to simplify the presentation, a write operation on a base register X, is denoted "change X". As any two consecutive write operations on a base bit X write different values, it follows that X behaves as regular register.

The 3 base safe bits used in the construction of the high level atomic register R are the following:

- REG: the safe bit that, intuitively, contains the value of the atomic bit that is constructed. It is written by the writer and read by the reader.
- WR: the safe bit written by the writer to pass control information to the reader.
- RR: the safe bit written by the reader to pass control information to the writer.

6.3.2. Handshaking mechanism and the write operation

As we saw in the previous section, the reader should inform the writer when it read a new value v in the implemented register. Otherwise, the uninformed writer may subsequently repeat the same digest of state transitions executing $R.write(v)$ so that the reader would be subject to new-old inversion. Therefore, whenever the writer is informed that a previously written value is read by the reader, it should change the execution so that critical digests are not repeated.

The basic idea of the construction is to use the control bits WR and RR to implement the *handshaking* mechanism. Intuitively, the writer informs the reader about a new value by changing the value of WR so that $WR \neq RR$. Respectively, the reader informs the writer that the new value is read by changing the value of RR so that $WR = RR$. With these conventions, we obtain the following handshaking protocol between the writer and the reader:

- After the writer has changed the value of the base register REG, if it observes $WR = RR$, it changes the value of WR.

  As we can see, setting the predicate $WR = RR$ equal to false is the way used by the writer to signal that a new value has been written in REG. The resulting is described in Figure 6.2.

```
operation $R.write(v)$: %Change the value of $R$ %
i   change $REG$;
ii  if $WR = RR$ then change $WR$; % Strive to establish $WR \neq RR$ %
    return ()
```

Figure 6.2.: The $R.write(v)$ operation

- Before reading REG, the reader changes the value of RR, if it observes that $WR \neq RR$. This signaling is used by the writer to update WR when it discovers that the previous value has been read.

As we are going to see in the rest of this chapter, the exchange of signals through WR and RR is also used by the reader to check if the value it has found in REG can be returned.

6.3.3. An incremental construction of the read operation

The reader's algorithm is much more involved than the writer's algorithm. To make it easier to understand, this section presents the reader's code in an incremental way, from simpler versions to more involved ones. In each stage of the construction, we exhibit scenarios in which a simpler version fails, which motivates a change of the protocol.

The construction: step 1 We start with the simplest construction in which the reader establishes $RR = WR$ and returns the value found in REG. (The line numbers are chosen to anticipate future modifications of the algorithm.)

```
3  **if** $WR \neq RR$ **then** change $RR$; % Strive to establish $WR = RR$ %
4  $val \leftarrow REG$;
5  *return* $(val)$
```

We can immediately see that this version does not really use the control information: the value returned by the read operation does not depend on the states of RR and WR. Consequently, this version is subject to new-old inversions: suppose that while the writer changes the value of REG from 0 to 1 (line ii in Figure 6.2), the reader performs two read operations. The first read returns 1 (the "new" value of R) and the second read returns 0 (the "old" value), i.e., we obtain a new-old inversion.

The construction: step 2 An obvious way to prevent the new-old inversion described in the previous step is to allow the reader to return the current value of REG only if it observes that the writer has updated WR to make $WR \neq RR$ since the previous read operation.

```
1   **if** $WR = RR$ **then** *return* $(val)$;
3'  change $RR$; % Strive to establish $WR = RR$ %
4   $val \leftarrow REG$;
5   *return* $(val)$
```

Here we assume that the local variable val initially contains the initial value of R (e.g., 0). Checking whether $WR \neq RR$ before changing RR in line 3′ looks unnecessary, since the reader does not touch the shared memory between reading WR in line 1 and in line 3, so we dropped it for the moment.

Unfortunately, we still have a problem with this construction. When a read is executed concurrently with a write, it may happen that the read returns a concurrently written value but a subsequent read finds $RR \neq WR$ and returns an old value found in REG.

Indeed, consider the following scenario (Figure 6.3):

1. $w_1 = R.write(1)$ changes REG and starts changing WR.
2. r_1 reads WR, finds $WR \neq RR$ and changes RR, reads REG and returns 1.
3. r_2 reads WR and still finds $WR \neq RR$ (new-old inversion on WR).
4. w_1 completes changing WR and returns.
5. $w_2 = R.write(0)$ starts changing REG.
6. r_2 changes RR (establishing that $RR \neq WR$ now), reads REG and returns 0.

7. r_3 reads WR, finds $WR \neq RR$, reads REG and returns 1 (new-old inversion on REG).

8. w_2 completes changing REG and returns.

In other words, we obtain a new-old inversion for read operations r_2 and r_3.

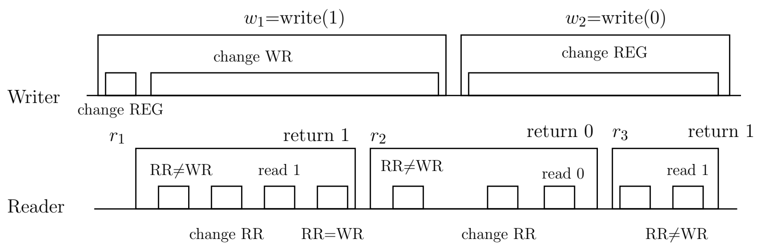


Figure 6.3.: Counter example to step 2 of the construction: new-old inversion for r_1 and r_2

The construction: step 3 The problem with the scenario above is that read operation r_2 changes RR while it is not necessary: it previously evaluated $WR \neq RR$ due to a new-old inversion on WR. Thus, when r_2 changes RR, it sets $WR \neq RR$ again. Thus, the subsequent read r_3 finds $WR \neq RR$ will be forced to return a value read in REG, and the value can be "old" due to the ongoing change in REG.

A naïve solution to this could be for the reader to check again if $WR \neq RR$ still holds before changing RR. By itself, this additional check will not change anything, since we could schedule this check performed by r_2 immediately after the first one and concurrently with w_1's change of WR. Thus, additionally, the reader may first read REG and only then check if the condition $WR \neq RR$ still holds and change RR if it does.

```
1  if $WR = RR$ then return $(val)$;
2′ $val \leftarrow REG$;
3  if $WR = RR$ then change $RR$;
5  return $(val)$
```

This way we fix the problem described in Figure 6.3 but face a new one. The value read in REG may get overly conservative in some cases. Consider, for example, the scenario in Figure 6.4. Here read operation r_2 evaluates $WR = RR$ and returns the old value 1, even though the most recently written value is actually 0. This is because, the preceding read operation r_1 changed RR to be equal to WR without noticing that REG was meanwhile changed

The construction: step 4 One solution to the problem exemplified in Figure 6.4 is, as put in the pseudocode below, to evaluate REG after changing RR and then check RR again. If the predicate $RR = WR$ does not hold after RR was changed and REG was read again, the reader returns the old (read in line 2) value of REG. Otherwise, the new (read in line 4) value is returned.

```
1  if $WR = RR$ then return $(val)$;
2  $aux \leftarrow REG$; % Conservative value %
3  if $WR = RR$ then change $RR$;
```

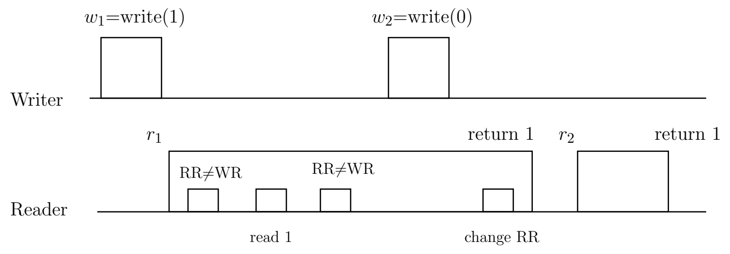


Figure 6.4.: Counter example to step 3 of the construction: r_2 returns an outdated value

```
4 $val \leftarrow REG$;
5 **if** $WR = RR$ **then** *return* $(val)$;
7 *return* $(aux)$
```

Unfortunately, there is still a problem here. The variable val evaluated in line 4 may be too conservative to be returned by a subsequent read operation that finds $RR = WR$ in line 1.

Again, suppose that $w_1 = R.write(1)$ is followed a concurrent execution of $r_1 = R.read()$ and $w_2 = R.write(0)$ as follows (Figure 6.5):

1. $w_1 = R.write(1)$ completes.

2. $w_2 = R.write(0)$ begins and starts changing REG from 1 to 0.

3. r_1 finds $WR \neq RR$, reads 0 from REG and stores it in aux (line 2), changes RR, reads 1 from REG and stores it in val (the write operation on REG performed by w_2 is still going on).

4. w_2 completes its write on REG, finds $RR = WR$ and starts changing WR.

5. r_1 finds $WR \neq RR$ (line 5), concludes that there is a concurrent write operation and returns the "conservative" value 0 (read in line 2).

6. $r_2 = R.read()$ begins, finds $RR = WR$ (the write operation on WR performed by w_2 is still going on), and returns 1 previously evaluated in line 4 of r_1.

That is, r_1 returned the new (concurrently written) value 0 while r_2 returned the old value 1.

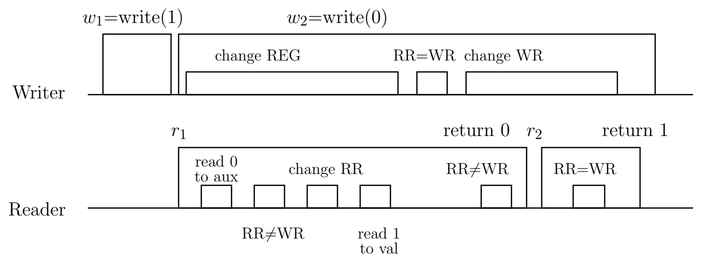


Figure 6.5.: Counter example to step 4 of the construction: new-old inversion for r_1 and r_2

The construction: last step The complete read algorithm is presented in Figure 6.6. As we saw in this chapter, safe base registers allow for a multitude of possible execution scenarios, so an intuitively correct implementation could be flawed because of an overlooked case. To be convinced that our construction is indeed correct, we provide a rigorous proof below.

```
operation $R.read()$:
  1  if $WR = RR$ then return $(val)$;
  2  $aux \leftarrow REG$;
  3  if $WR \neq RR$ then change $RR$;
  4  $val \leftarrow REG$;
  5  if $WR = RR$ then return $(val)$;
  6  $val \leftarrow REG$;
  7  return $(aux)$
```

Figure 6.6.: The $R.read()$ operation

6.3.4. Proof of the construction

Theorem 17 *Let H be an execution history of the 1W1R register R constructed by the algorithm in Figures 6.2 and 6.6. Then H is linearizable.*

Proof Let H be an execution history. By Theorem 5, to show that H is linearizable (atomic), it is sufficient to show that there exists a reading function π satisfying the assertions $A0$, $A1$ and $A2$.

In order to distinguish the operations $R.read()$ and $R.write(v)$, denoted by r and w, from the read and write operations on the base registers (e.g., "change RR", "$aux \leftarrow REG$", etc.), the latter ones are called *actions*. The corresponding execution containing, additionally, the invocation and response events on base registers is denoted L. Let $\rightarrow_L$ denote the corresponding partial relation on the actions.

Moreover, r being a read operation and loc the local variable (aux or val) whose value is returned by r (in line 1, 5 or 7), ρ_r denotes the last read action "$loc \leftarrow REG$" executed before r returns:

- If r returns in line 7, ρ_r is the read action "$aux \leftarrow REG$" executed in line 2 of r,
- If r returns in line 5, ρ_r is is the read action "$val \leftarrow REG$" executed in line 4 of r, and finally
- If r returns in line 1, ρ_r is is the read action "$val \leftarrow REG$" executed in line 4 or 6 of some previous read operation.

Let ϕ be any regular reading function on REG. Thus, for each read action ρ_r we can define the corresponding write action $\phi(\rho_r)$ that writes the value returned by r. The write operation that contains $\phi(\rho_r)$ determines $\pi(r)$. If there is no such write operation, i.e., ρ_r returns the initial value of REG, we assume that $\pi(r)$ is the (imaginary) initial write operation that writes the initial value and precedes all actions in H.

Proof of $A0$. Let r be a complete read operation in H. By the definition of π, the invocation of the write action $\phi(\rho_r)$ occurs before the response of ρ_r and, thus, the response of r in L, i.e., $inv[\pi(\rho_r)] <_L resp[r]$. Thus, $inv[\pi(r)] <_L inv[\pi(\rho_r)] <_L resp[r]$ and $\neg(resp[r] <_L inv[\pi(r)])$.

By contradiction, suppose that $A0$ is violated, i.e., $r \rightarrow_H \pi(r)$. Thus, $resp[r] <_L inv[\pi(\rho_r)])$—a contradiction.

Proof of $A1$. Since there is only one writer, all writes are totally ordered and $w \rightarrow_H \pi(r)$ is equivalent to $\neg(\pi(r) \rightarrow_H w)$.

By contradiction, suppose that there is a write operation w such that $\pi(r) \rightarrow_H w \rightarrow_H r$. If there are several such write operations, let w be the last one before r, i.e., $\nexists\, w'$: $w \rightarrow_H w' \rightarrow_H r$.
We first claim that, in such a context, ρ_r cannot be a read action of the read operation r (i.e., $\rho_r \notin r$).
Proof of the claim. Recall that $\phi(\rho_r) \in \pi(r)$ (by definition). Let ω be the "change REG" action of the operation w ($\omega \in w$). By the case assumption, we obtain $\phi(\rho_r) \rightarrow_L \omega$. By the definition of $\phi(\rho_r)$, we have $\neg(\rho_r \rightarrow_L \phi(\rho_r))$ and, thus, $\neg(\omega \rightarrow_L \rho_r)$. Therefore, $inv[\rho_r] <_L resp[\omega]$. As $\omega \in w$ and $w \rightarrow_H r$, we have $inv[\rho_r] <_L resp[w] <_L inv[r]$. As ρ_r started before r, and both are executed by the same process, we have $\rho_r \notin r$. *End of the proof of the claim.*

Since $\rho_r \notin r$, by the algorithm in Figure 6.6, the read operation r returns a value in line 1, which means that it has previously seen $WR = RR$. On the other hand, after the writer has executed ω within $\pi(r)$, it read RR in order to set WR different from RR if they were seen equal. As $w \rightarrow_H r$ and $\nexists\, w'$: $w \rightarrow_H w' \rightarrow_H r$ (assumption), it follows that RR has been modified by a read operation in line 3 *before* the read operation r starts but *after or concurrently with* the read action on RR performed by w. Let r' be that read operation; as there is a single process executing $R.read()$, we have $r' \rightarrow_H r$.
Now we claim that $\rho_r \notin r'$.
Proof of the claim: Let r'' be the read operation that contains ρ_r. We show that $r'' \neq r'$. We observe that (Figure 6.7):

- If r'' updates RR, it does it in line 3, i.e., before executing ρ_r (in line 4 or 6),
- $inv[\rho_r] <_L resp[\omega]$ (since ϕ is a regular reading function and $\phi(\rho_r)$ precedes ω); the relation between $\phi(\rho_r)$ precedes ω is indicated by a dotted arrow in Figure 6.7),
- w reads RR after having executed ω (code of the write operation).

It follows from these observations that if r'' writes into RR, then it completes the write before w starts reading RR. But r' writes to RR either after or concurrently with the read of RR performed within w. Therefore, $r'' \neq r'$ and, thus, $\rho_r \notin r'$. *End of the proof of the claim.*

But since the reader modifies RR within r', it also executes line 4 of r' ($val \leftarrow REG$) before executing r (this follows from the code of the read operation). But, as $\rho_r \notin r'$, this read of REG action within r' contradicts the definition of ρ_r (according to which ρ_r is the last action "$val \leftarrow REG$" executed before r starts), which completes the proof of the assertion $A1$.

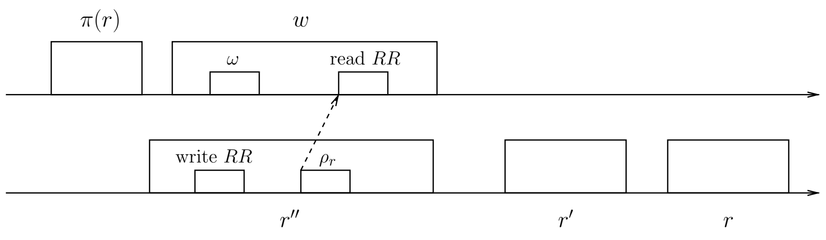


Figure 6.7.: ρ_r belongs neither to r nor to r'

Proof of $A2$. By contradiction, suppose that there exist $r1$ and $r2$, two complete read operations in H, such that $r1 \rightarrow_H r2$ and $\pi(r2) \rightarrow_H \pi(r1)$. Without loss of generality, we assume that if $r1$ returns at line 1, then ρ_{r1} is the read action in line 6 in the immediately preceding read operation. Since $\pi(r2) \neq \pi(r1)$, we have $\rho_{r1} \neq \rho_{r2}$. Thus, either $\rho_{r1} \rightarrow_L \rho_{r2}$ or $\rho_{r2} \rightarrow_L \rho_{r1}$.

- $\rho_{r2} \rightarrow_L \rho_{r1}$.
  As ρ_{r1} precedes or belongs to $r1$, and $r1 \rightarrow_H r2$, we have $resp[\rho_{r1}] <_L inv[r2]$. Combined with the case assumption, the assertion implies $\rho_{r2} \rightarrow_L \rho_{r1} \rightarrow_L r2$, which contradicts the fact that ρ_{r2} is the last "$loc \leftarrow REG$" action executed before $r2$ started, where loc is val or aux. So, the case $\rho_{r2} \rightarrow_L \rho_{r1}$ is not possible.

- $\rho_{r1} \rightarrow_L \rho_{r2}$.
  By definition $\phi(\rho_{r1}) \in \pi(r1)$ and $\phi(\rho_{r2}) \in \pi(r2)$. As $\pi(r2) \rightarrow_H \pi(r1)$, we have $\phi(\rho_{r2}) \rightarrow_L \phi(\rho_{r1})$.

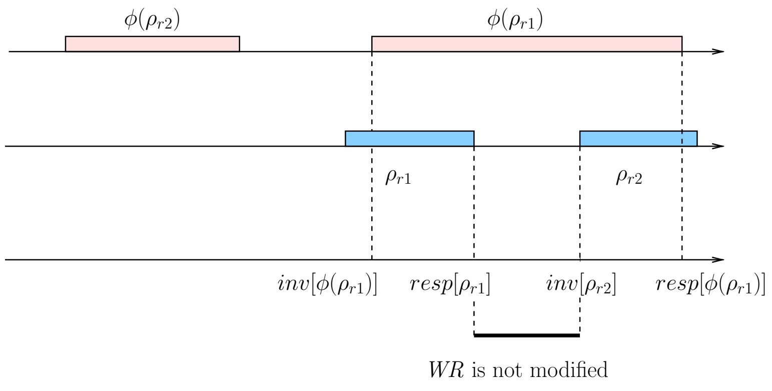


Figure 6.8.: A new-old inversion on the regular register REG

Thus, we have $\phi(\rho_{r2}) \rightarrow_L \phi(\rho_{r1})$ and $\rho_{r1} \rightarrow_L \rho_{r2}$ (Figure 6.8) which implies a new-old inversion on the base regular register REG. But since ϕ is a regular reading function on REG, we have $\neg(\rho_{r1} \rightarrow_L \phi(rho_{r1})$ and $\neg(\phi(\rho_{r1}) \rightarrow_L \rho_{r2})$. Thus, both ρ_{r1} and ρ_{r2} have to overlap $\pi(\rho_{r1})$ (Figure 6.8): $inv[\phi(\rho_{r1})] <_L resp[\rho_1]$ and $inv[\rho_2] <_L resp[\phi(\rho_{r1})]$. As $\phi(\rho_{r1})$ is a base action that updates REG, and as REG and WR are both updated by the writer, the "value" of the base register WR does not change while the writer is updating REG or, more formally:

Property P: all read actions on WR performed between $resp[\rho_{r1}]$ and $inv[\rho_{r2}]$ return the same value.

We consider three cases according to the line at which $r1$ returns.

- $r1$ returns in line 7.
  Then ρ_{r1} is "$aux \leftarrow REG$" in line 2 of $r1$. We have the following:
  - Since $\rho_{r1} \rightarrow_L \rho_{r2}$ and $r1$ returns in line 7, ρ_{r2} can only be the read in line 6 of $r1$ or a later read action.
  - After having performed ρ_{r1}, $r1$ reads WR and if $WR \neq RR$, it sets $RR = WR$ in line 3. But $r1$ returns in line 7, after having seen RR different from WR in line 5 (otherwise, it would have returned in line 5). Thus, $r1$ reads different values of WR after ρ_{r1} (line 2 of $r1$) and before ρ_{r2} (line 6 of $r1$ or later). This contradicts property P above.

- $r1$ returns in line 5.
  Then, ρ_{r1} is "$val \leftarrow REG$" in line 4 of $r1$, and $r1$ sees $RR = WR$ in line 5. Since $\rho_{r1} \rightarrow_L \rho_{r2}$, $r2$ does not return in line 1. Indeed, if $r2$ returns in line 1, the property P implies that the last read on REG preceding line 1 of $r2$ is line 4 of $r1$, i.e., $\rho_{r1} = \rho_{r2}$. Thus, $r2$ sees $RR \neq WR$ in line 1, before performing ρ_{r2} is in line 2 or line 4 of $r2$. But $r1$ has seen $WR = RR$ in line 5, after having performed ρ_{r1} in line 4—a contradiction with property P.

- $r1$ returns in line 1.
  In that case, ρ_{r1} is line 4 or line 6 of the read operation that precedes $r1$. Again, since $\rho_{r1} \rightarrow_L \rho_{r2}$, $r2$ does not return in line 1, from which we conclude that, before performing ρ_{r2}, $r2$ sees $RR \neq WR$ in line 1. On the other hand, $r1$ sees $RR = WR$ in line 1 after having performed ρ_{r1} which contradicts property P and concludes the proof.

Thus, π is an atomic reading function. $\square_{Theorem\ 17}$

6.3.5. Cost of the algorithms

The cost of the $R.read()$ and $R.write(v)$ operations is measured by the the maximal and minimal numbers of accesses to the base registers. Let us remind that the writer (resp., reader) does not read WR (resp., RR) as it keeps a local copy of that register.

- $R.write(v)$: maximal cost: 3; minimal cost: 2.
- $R.read()$: maximal cost: 7; minimal cost: 1.

The minimal cost is realized when the same type of operation (i.e., read or write) is repeatedly executed while the operation of the other type is not invoked.

Notice we have assumed that if $R.write(v)$ and $R.write(v')$ are two consecutive write operations, we have $v \neq v'$. If the user issues two consecutive write operations with the same argument, the cost of the second one is 0, as it is skipped and consequently there is no accesses to base registers.

6.4. Bibliographic notes

Lamport stated the problem of implementing atomic abstractions from weaker ones [49]. One of the algorithms can be used to implement an unbounded atomic registers using unbounded regular ones. The direct bounded construction of a binary atomic shared register discussed in this chapter was proposed by Tromp [67, 68].

7. Atomic multivalued register construction

In Chapter 5, we described an implementation of an atomic 1WNR register from regular ones that uses sequence numbers growing without bound and, thus, must assume base registers of unbounded capacity. In this chapter, we propose a *bounded* solution. But let us first recall a few related constructions we discussed earlier.

7.1. From single-reader regular to multi-reader atomic

In Chapter 6, we discussed how to construct an atomic *bit* from only three safe bits. One of the bits is used for storing the value itself, and the other two are used for exchanging control signals between the writer and the reader. In the one-reader case, we can turn a series of atomic 1W1R bits into an atomic *bounded multi-valued* register using the simple transformation algorithm in Section 4.8. But how do we construct a *multi-reader* multi-valued atomic register?

It is straightforward to get a *regular* bounded multi-valued multi-reader register from single-reader ones (recall the algorithms in Section 4.4). This chapter describes how to construct an *atomic* one.

We begin with describing a simpler algorithm that, in addition to regular registers used to store the written value itself, employs an atomic bit used for transmitting control signals from the writer to the readers.

7.2. Using an atomic control bit

The construction of a multi-reader register using two regular registers REG_1 and REG_2 and an atomic bit $WFLAG$ is given in Figure 7.1.

```
 operation $R.write(v)$:
(1) $WFLAG \leftarrow true$;
(2) $REG_1 \leftarrow v$;
(3) $WFLAG \leftarrow false$;
(4) $REG_2 \leftarrow v$;

 operation $R.read()$:
(5) $val \leftarrow REG_1$;
(6) if $\neg WFLAG$ then return($val$);
(7) $val \leftarrow REG_2$;
(8) return ($val$)
```

Figure 7.1.: From regular registers and an atomic control bit to an atomic register.

In the algorithm, the value is written twice: first in REG_1 and then in REG_2. Before writing to REG_1, the writer sets WFLAG to $true$ to signal to the readers the beginning of a new write operation. After writing to REG_1, the writer sets WFLAG back to $false$.

A read operation reads REG_1 and then checks $WFLAG$. If $WFLAG$ contained $false$, then the process returns the value previously read in REG_1. If $WFLAG$ contained $true$, then the process reads and returns the value in REG_2.

Intuitively, $\mathit{WFLAG} = \mathit{true}$ means that there is a possibility that the value found earlier in REG_1 is written by a concurrent write operation and, therefore, a subsequent read operation might find the older value in REG_1, due to new-old inversion on REG_1. To prevent, new-old inversion on the implemented register, it is therefore necessary to return a more conservative value read in REG_2.

Theorem 18 *The algorithm in Figure 7.1 implements a 1WMR atomic register using one 1WMR atomic bit and two 1WMR regular registers.*

Proof Let H be a history of the algorithm in Figure 7.1, and let L be the corresponding execution. Let π be any regular reading function defined on read operations on REG_1 or REG_2. We extend π to the high-level read operations on the implemented register R as follows. For each high-level read r returning the value found by a read operation ρ in REG_1 or REG_2 (in lines 5 or 7), let $\pi(r)$ be the high-level write operation w that contains $\pi(\rho)$.

It is immediate from the construction that the resulting extension of π on high-level read operations is regular. Indeed, the interval of every such $\pi(\rho)$ belongs to the interval of w. Thus, $\rho \not\to_L \pi(\rho)$ implies $r \not\to_H \pi(r)$, i.e., $A0$ is satisfied. Additionally, since every complete write operation contains writes on both REG_1 and REG_2, $A1$ satisfied by π defined over reads of REG_1 and REG_2 implies that for any w and r, we cannot have $\pi(r) \to_H w \to_H r$, i.e., $A1$ is satisfied.

Now we are going to prove $A2$. By contradiction, suppose that for two high-level operations r_1 and r_2, we have $r_1 \to_H r_2$ and $\pi(r_2) \to_H \pi(r_1)$. For $i = 1, 2$, let ρ_i be the read operation on REG_1 or REG_2 that was used by r_i to evaluate the returned value. Clearly, $\rho_1 \to_L \rho_2$.

The following cases are possible:

(1) Both ρ_1 and ρ_2 read REG_1.

By property $A1$ of regular functions, $\pi(\rho_1) \not\to_L \rho_2$: otherwise we would have $\pi(\rho_2) \to_L \pi(\rho_1) \to_L \rho_2$, i.e., ρ_2 would return an "overwritten" value. By property $A0$, $\rho_1 \not\to_L \pi(\rho_1)$. Thus, given that $\rho_1 \to_L \rho_2$, $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2.

By the algorithm, just before writing to REG_1 in $\pi(\rho_1)$, operation $\pi(r_1)$ has set WFLAG to true. Since $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2, no write on WFLAG took place in the interval between the response of ρ_1 and the invocation of ρ_2. Notice that r_1 checks WFLAG during this interval and, thus, true was the last written value on WFLAG when it is read within r_1. Thus, after having read REG_1, r_1 must have found true in WFLAG and returned the value read in REG_2—a contradiction with the assumption that the value read in REG_1 is returned by r_1.

(2) Both ρ_1 and ρ_2 read REG_2.

Similarly, using $A0$ and $A1$, we derive that $\pi(\rho_1)$, updating REG_2, is concurrent with both ρ_1 and ρ_2. By the algorithm, just before writing to REG_2, $\pi(r_1)$ has set WFLAG to false. Thus, before reading REG_2, r_2 must have read false in WFLAG and returned the value read in REG_1—a contradiction with the assumption that the value read in REG_2 is returned by r_2.

(3) ρ_1 reads REG_2 and ρ_2 reads REG_1.

In $\pi(r_1)$, $\pi(\rho_1)$ is preceded by a write wr_1 on REG_1: $wr_1 \to_L \pi(\rho_1)$. By $A0$, $\rho_1 \not\to_L \pi(\rho_1)$. Now relations $wr_1 \to_L \pi(\rho_1)$, $\rho_1 \not\to_L \pi(\rho_1)$, and $\rho_1 \to_L \rho_2$ imply $wr_1 \to_L \rho_2$.

But, by our assumption, $\pi(r_2) \to_H \pi(r_1)$ and, thus, $\pi(\rho_2) \to_L wr_1$, which, together with $wr_1 \to_L \rho_2$, implies $\pi(\rho_2) \to_L wr_1 \to_L \rho_2$, violating $A1$—a contradiction.

(4) ρ_1 reads REG_1 and ρ_2 reads REG_2.

By the algorithm, after ρ_1 has returned, r_1 found false in WFLAG. After that r_2 read REG_1, found true in WFLAG, and then read and returned the value in REG_2. Let rf_1 and rf_2 be the read operations of WFLAG performed within r_1 and r_2, respectively. Thus, $\rho_1 \rightarrow_L \mathit{rf}_1 \rightarrow_L \mathit{rf}_2 \rightarrow_L \rho_2$.

Since WFLAG is atomic, there must be a write operation wf on WFLAG changing its value from false to true (line 1) that is linearized between linearizations of rf_1 and rf_2 and, thus, $\mathit{wf} \not\rightarrow_L \mathit{rf}_1$ and $\mathit{rf}_2 \not\rightarrow_L \mathit{wf}$. Let wr_1 and wr_2 be the write operations on, respectively, REG_1 and REG_2 that immediately precede wf. (Recall that wr_1 and wr_2 can belong to the initializing write operation on R.)

Now we derive that $\pi(\rho_1)$ must be wr_1 or an earlier write on REG_1. Otherwise, we would get $\mathit{wf} \rightarrow_L \pi(\rho_1)$ which, combined with $\rho_1 \rightarrow_L \mathit{rf}_1$ and $\mathit{wf} \not\rightarrow_L \mathit{rf}_1$, implies that $\rho_1 \rightarrow_L \pi(\rho_1)$—a violation of $A0$.

On the other hand, by $A1$, there does not exist wr, a write operation on REG_2, such that $\pi(\rho_2) \rightarrow_L \mathit{wr} \rightarrow_L \rho_2$.

Similarly, $\pi(\rho_2)$ must be wr_2 or a later write on REG_2. Otherwise, we would get $\pi(\rho_2) \rightarrow_L \mathit{wr}_2$. But $\mathit{wr}_2 \rightarrow_L \mathit{wf}$, $\mathit{rf}_2 \not\rightarrow_L \mathit{wf}$ and $\mathit{rf}_2 \rightarrow_L \rho_2$ imply $\mathit{wr}_2 \rightarrow_L \rho_2$. Thus, $\pi(\rho_2) \rightarrow_L \mathit{wr}_2 \rightarrow_L \rho_2$—a violation of $A1$.

Therefore, $\pi(\rho_1) \rightarrow_L \pi(\rho_2)$ and, thus, $\pi(r_1) = \pi(r_2)$ or $\pi(r_1) \rightarrow_H \pi(r_2)$—a contradiction.

Hence, π satisfied $A2$ and the algorithm indeed implements an atomic register. $\Box_{\mathit{Theorem}\ 18}$

Notice that we only used the fact that WFLAG is atomic in case (4). By replacing WFLAG with a regular register, or a set of registers providing the functionality of one regular register, we would maintain atomicity in cases (1)-(3). However, as we will see in the next section, taking care of case (4) incurs nontrivial changes in processing the remaining cases.

7.3. The algorithm

The bounded algorithm transforming regular multi-valued multi-reader registers into an atomic one is presented in Figure 7.2. Notice that we replaced the atomic control bit WFLAG in the algorithm in Figure 7.1 with several regular registers of bounded capacity:

- $\mathit{LEVEL} = 0, 1, 2$: a ternary regular register used by the writer to signal to the readers at which “stage of writing” it currently is.
- $\mathit{FC}[1, \dots, n]$: an array of regular binary registers, each $\mathit{FC}[i]$ is written by reader p_i and by read by the other readers.
- $\mathit{RC}[1, \dots, n]$: an array of regular binary registers, each $\mathit{RC}[i]$ is written by reader p_i and read by the writer and other readers.
- $\mathit{WC}[1, \dots, n]$: an array of regular binary registers, written by the writer and read by the readers.

Intuitively, $\mathit{LEVEL} = 1$ corresponds to $\mathit{WFLAG} = \mathit{true}$, and $\mathit{LEVEL} = 2$ and $\mathit{LEVEL} = 0$ correspond to $\mathit{WFLAG} = \mathit{false}$ in the algorithm in Figure 7.1. But LEVEL is a regular register now. Hence, to handle the possible new-old inversion on LEVEL, the readers exchange information with each other using the array $\mathit{FC}[1, \dots, n]$ and with the writer using the arrays $\mathit{RC}[1, \dots, n]$ and $\mathit{WC}[1, \dots, n]$.

```
operation $R.write(v)$:
(1)  $LEVEL \leftarrow 1$;
(2)  $REG_1 \leftarrow v$;
(3)  $LEVEL \leftarrow 2$;
(4)  $LEVEL \leftarrow 0$;
(5)  $REG_2 \leftarrow v$;
(6)  for $j = 1, \ldots, n$ do
(7)      $lr \leftarrow RC[j]$;
(8)      $WC[j] \leftarrow \neg lr$;

operation $R.read()$ (code for reader $p_i$):
(9)  $val \leftarrow REG_1$;
(10) $lw \leftarrow WC[i]$;
(11) if $lw \neq RC[i]$ then
(12)     $FC[i] \leftarrow false$;
(13)     $RC[i] \leftarrow lw$;
(14) case $LEVEL$ do
(15) 0:  return($val$);
(16) 2:  $FC[i] \leftarrow true$; return($val$);
(17) 1:  for $j = 1, \ldots, n$ do
(18)         $lr \leftarrow RC[j]$;
(19)         $lf \leftarrow FC[j]$;
(20)         $lw \leftarrow WC[j]$;
(21)         if $(lr = lw) \wedge lf$ then
(22)             $FC[i] \leftarrow true$;
(23)             return ($val$);
(24)     $val \leftarrow REG_2$;
(25)     return($val$);
```

Figure 7.2.: From bounded regular registers to a bounded atomic register.

Theorem 19 *The algorithm in Figure 7.2 implements a 1WMR atomic register using 1WMR regular registers.*

Proof Consider a history H and the corresponding execution L of the algorithm in Figure 7.2. As in the proof of Theorem 18, we take any reading function π acting over read operations on base regular registers, and then extend it to high-level read operations on the implemented register R as follows. For each complete high-level operation r returning the value read by an operation ρ in REG_1 (line 9) or REG_2 (line 24), let $\pi(r)$ be the high-level write operation w that contains $\pi(\rho)$. It is immediate that π, as a function on high-level reads, is regular.

Now assume, by contradiction, that π is not atomic, i.e., there exist two high-level operations r_1 and r_2, such that $r_1 \rightarrow_H r_2$ and $\pi(r_2) \rightarrow_H \pi(r_1)$. For $i = 1, 2$, let ρ_i be the read operation on REG_1 or REG_2 that was used by r_i to evaluate the returned value.

For brevity, we introduce the following notation:

- $w_1 = \pi(\rho_1)$ and $w_2 = \pi(\rho_2)$;
- $wr_{i,j}$ denotes the write to REG_j performed within w_i ($i = 1, 2$, $j = 1, 2$), if any;
- $rr_{i,j}$ denotes the read of REG_j performed within r_i ($i = 1, 2$, $j = 1, 2$);
- $wl_{i,j}$ denotes j-th write to $LEVEL$ performed within w_i ($i = 1, 2$, $j = 1, 2, 3$), if any; note that $wl_{i,j}$ writes the value $j \mod 3$;
- rl_i denotes the read operations on $LEVEL$, performed within r_i ($i = 1, 2$).

Since every complete high-level write operation contains writes on both REG_1 and REG_2, it follows that w_2 immediately precedes w_1. Otherwise, regardless of which register REG_i ($i = 1, 2$) is read by ρ_2, we would have a write wr on REG_i such that $\pi(\rho_2) \rightarrow_L wr \rightarrow_L \pi(\rho_1)$ which, combined with $\rho_1 \not\rightarrow_L \pi(\rho_1)$ and $\rho_1 \rightarrow_L \rho_2$ (our initial assumption), would imply $\pi(\rho_2) \rightarrow_L wr \rightarrow_L \rho_2$—a violation of $A1$ for ρ_2.

As in the proof of Theorem 18, we now should consider the four following cases:

(1) ρ_1 reads REG_2 and ρ_2 reads REG_1.

Since $w_2 \rightarrow_H w_1$, we have $\pi(\rho_2) \rightarrow_L wr_{1,1} \rightarrow_L \pi(\rho_1)$. Now, by $A0$, $\rho_1 \not\rightarrow_L \pi(\rho_1)$, which, together with $\rho_1 \rightarrow_L \rho_2$, implies $\pi(\rho_2) \rightarrow_L wr_{1,1} \rightarrow_L \rho_2$—a violation of $A1$ for ρ_2.

(2) Both ρ_1 and ρ_2 read REG_2.

Properties $A0$ and $A1$ imply that $\pi(\rho_1) \not\rightarrow_L \rho_2$ and $\rho_1 \not\rightarrow_L \pi(\rho_1)$, i.e., $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2. Thus, no write on $LEVEL$ takes place between the response of ρ_1 and the invocation ρ_2. By the algorithm, immediately before updating REG_2, w_1 writes 0 to $LEVEL$. Thus, before reading REG_2, r_2 must have read 0 in $LEVEL$ and return the value read in REG_1—a contradiction.

(3) ρ_1 reads REG_1 and ρ_2 reads REG_2.

Just before updating REG_1 in $\pi(\rho_1)$, w_1 writes 1 to $LEVEL$ in operation $wl_{1,1}$, thus, $wl_{1,1} \rightarrow_L \pi(\rho_1)$, $\rho_1 \rightarrow_L rl_1$, and $\rho_1 \not\rightarrow_L \pi(\rho_1)$ (property $A0$) imply $wl_{1,1} \rightarrow_L rl_1 \rightarrow_L rl_2$.

By the algorithm, r_2 must have read 1 in $LEVEL$. Suppose that $wl_{1,1} \neq \pi(rl_2)$, i.e., rl_2 reads 1 written to $LEVEL$ by another write operation wl. Since $wl_{1,1} \rightarrow_L rl_2$, property $A1$ for rl_2 implies $wl_{1,1} \rightarrow_L wl$. By the algorithm, since wl writes 1, we have $wl_{1,2} \rightarrow_L wl$. But $\pi(\rho_2) \rightarrow_L wr_{1,2}$ (since $w_2 \rightarrow_H w_1$), $rl_2 \not\rightarrow_L wl$ ($A0$ for rl_2), and $rl_2 \rightarrow_L \rho_2$ (by the algorithm). Therefore, $\pi(\rho_2) \rightarrow_L wr_{1,2} \rightarrow_L \rho_2$—a violation of $A1$ for ρ_2. Thus, $\pi(rl_2) = wl_{1,1}$.

Since $rl_1 \rightarrow_L rl_2$ (by the assumption), $wl_{1,2} \not\rightarrow_L rl_2$ ($A1$ for rl_2), and $wl_{1,2} \rightarrow_L wl_{1,3}$ (by the algorithm), we have $rl_1 \rightarrow_L wl_{1,3}$. Also, since $wl_{1,1} \rightarrow_L wr_{1,1}$, $\rho_1 \rightarrow_L rl_1$ (by the algorithm), and $\rho_1 \not\rightarrow_L wr_{1,1}$ ($A0$ for ρ_1), we have $wl_{1,1} \rightarrow_L rl_1$. Furthermore, $rl_1 \rightarrow_L wl_{1,3}$: otherwise, $wl_{1,2} \rightarrow_L wl_{1,3}$ and $rl_1 \rightarrow_L rl_2$ would imply $wl_{1,1} \rightarrow_L wl_{1,2} \rightarrow_L rl_2$—a violation of $A1$ for rl_2.

Thus, by the algorithm, rl_1 reads either 1 written by $wl_{1,1}$ or 2 written by $wl_{1,2}$. In both cases, r_1 (executed, e.g., by reader p_i) sets $FC[i]$ to *true* before returning the value read by ρ_1 (in lines 16 or 22).

Since ρ_2 reads REG_2, we have $wr_{1,2} \not\rightarrow_L \rho_2$, otherwise we would violate $A1$ by having $\pi(\rho_2) \rightarrow_L wr_{1,2} \rightarrow_L \rho_2$. Thus, $\rho_1 \not\rightarrow_L \pi(\rho_1)$ and $wr_{1,2} \not\rightarrow_L \rho_2$ imply that the writer performs no updates on registers $WC[i]$ in the interval between the response of ρ_1 and before r_2 finishes reading $WC[i]$. Note that, within this interval, r_1 makes sure that $RC[i] = WC[i]$ and then sets $FC[i]$ to *true*.

Any subsequent operation rw performed by p_i writing *false* in $FC[i]$ or modifying $RC[i]$ can only take place if p_i previously finds out that $RC[i] \neq WC[i]$ (line 11), which cannot take place before a write on $WC[i]$ performed by the writer which, by the algorithm, must succeed $wr_{1,2}$: indeed, after r_1 ensures $RC[i] = WC[i]$ and sets $FC[i]$ to *true* and before it sets $FC[i]$ to *false* and modifies $RC[i]$ (lines 12 and 13), the writer must modify $WC[i]$ which can only happen after $wr_{1,2}$.

Thus, reads of $RC[i]$ and $FC[i]$ performed by r_2 precede rw, and the values read by r_2 satisfy $RC[i] = WC[i]$ and $FC[i] = \mathit{true}$ (Figure 7.3). By the algorithm, r_2 must then return the value of REG_1—a contradiction.

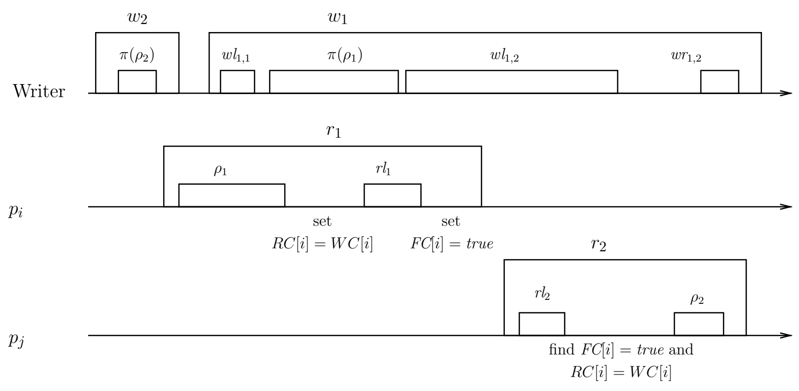


Figure 7.3.: An execution in case (3): r_2 finds out that $RC[i] = WC[i]$, so it cannot return the value read in REG_2.

(4) Both ρ_1 and ρ_2 read REG_1.

By $A0$, $\rho_1 \not\to_L \pi(\rho_1)$ and by $A1$, $\pi(\rho_1) \not\to_L \rho_2$, i.e., $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2.

Hence, $\pi(rl_1) = wl_{1,1}$, i.e., r_1 reads 1 in $LEVEL$, and then returns the value of REG_1 in line 23 before the response of $\pi(\rho_1)$.

We say that a read operation r_k *finishes its check-forwarding* when it executes the last read operation on some $WC[j]$ in line 20 before exiting the *for* loop starting in line 17. For any operation op, we write $cf_k \to_L op$ if r_k finishes its check-forwarding before the invocation of op.

Consider now any (high-level) read operation r_k returning in lines 23 or 25 such that:

(1) $rl_k \not\to_L wl_{1,1}$, and

(2) $cf_k \to_L wl_{1,2}$.

Note that r_1 satisfies these conditions. We establish a contradiction by showing that no such r_k can return in line 23.

For read operations r_ℓ and r_m, we say that r_ℓ *finishes check-forwarding before* r_m, and we write $cf_\ell \to_L cf_m$, if the last read operation of the check-forwarding phase of r_ℓ precedes the last read operation of the check-forwarding phase of r_m.

By contradiction, assume that there is a non-empty set R of read operations satisfying conditions (1) and (2) above that return in line 23. Without loss of generality, let r_k be any operation in R, such that no other operation in R finishes its check-forwarding before r_k.

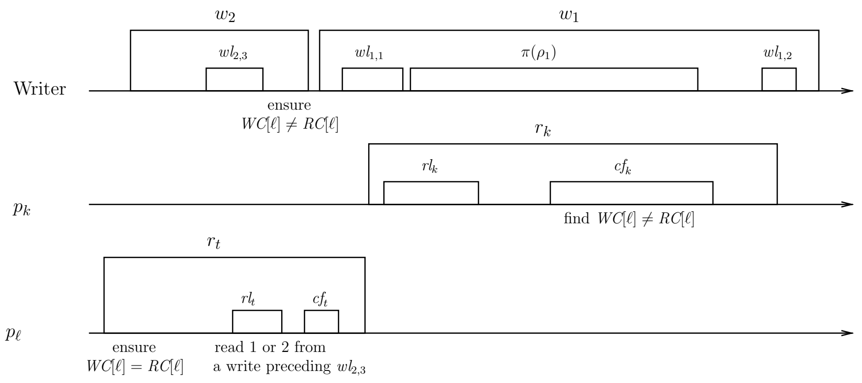
By the algorithm, before returning in line 23, r_k finds out that, for some reader p_ℓ, $FC[\ell] = true$ and $WC[\ell] = RC[\ell]$. Let r_t be the read operation performed by p_ℓ that, according to the reading function π, wrote this value in $FC[\ell]$. Let rf denote the read operation on $FC[\ell]$ performed within r_k (line 19), and let wf denote the write operation on $FC[\ell]$ performed within r_t (lines 16 or 22), i.e., $\pi(rf) = wf$. By the algorithm, before executing wf, r_t read 1 or 2 in $LEVEL$.

First we are going to show that r_t reads the value written in $LEVEL$ by a write operation that precedes w_1. Since $rf \to_L wl_{1,2}$ ($r_k \in R$ and the check-forwarding phases of reads in R satisfy condition (2) above), $rl_t \to_L wf$ (by the algorithm), and $rf \not\to_L wf$ ($A0$ for rf), we have $rl_t \to_L wl_{1,2}$ that is rl_t returns the value written by $wl_{1,1}$ or an earlier write.

Suppose, by contradiction, that $\pi(rl_t) = wl_{1,1}$, i.e., rl_t returns 1 written by $wl_{1,1}$. By $A0$, we have $rl_t \not\to_L wl_{1,1}$. Note that the fact that the last read operation of cf_k succeeds rf, $cf_t \to_L wf$ (by the algorithm), and $rf \not\to_L wf$ ($A0$ for rf) imply $cf_t \to_L cf_k$. But $cf_t \to_L wf$ and $rf \to_L wl_{1,2}$ imply $cf_t \to_L wl_{1,2}$, i.e., r_t satisfies conditions (1) and (2), while $cf_t \to_L cf_k$—a contradiction with the definition of r_k.

Hence, rl_t returns a value written by a write operation on $LEVEL$ preceding w_1. Since r_t modified $FC[\ell]$, rl_t must have returned 1 or 2, and $wl_{2,3} \not\to_L rl_t$ (otherwise, the only value that rl_t can return is 0). Note that, by the algorithm, any subsequent read operation by p_ℓ must set $FC[\ell]$ to $false$ (line 12) before modifying $RC[\ell]$ (line 13). Since r_k first reads $RC[\ell]$ and then reads $true$ in $FC[\ell]$ written by wf, the value of $RC[\ell]$ read by r_k must then be the value that r_t has "ensured", i.e., written or read in its last operation on $RC[\ell]$. Also, w_2 reads $RC[\ell]$ after the invocation of rl_t and before r_k read $RC[\ell]$, therefore it must read the same value of $RC[\ell]$.

Recall that after executing $wl_{2,3}$, w_2 ensures that $WC[\ell] \neq RC[\ell]$. Since, no succeeding update on $WC[\ell]$ takes place before r_k finishes its check-forwarding, the value of $WC[\ell]$ read by r_k must be the value that w_2 has previously ensured (Figure 7.4).



w_2
w_1
$wl_{2,3}$
$wl_{1,1}$
$\pi(\rho_1)$
$wl_{1,2}$
Writer
ensure $WC[\ell] \neq RC[\ell]$
r_k
rl_k
cf_k
p_k
find $WC[\ell] \neq RC[\ell]$
r_t
rl_t
cf_t
p_ℓ
ensure $WC[\ell] = RC[\ell]$
read 1 or 2 from a write preceding $wl_{2,3}$

Figure 7.4.: An execution in case (4): r_k finds out that $RC[\ell] \neq WC[\ell]$, so it cannot return the value read in REG_1.

Thus, r_k will find $WC[\ell] \neq RC[\ell]$—a contradiction with the assumption that r_k returns line 23 after finding out that $FC[\ell] = true$ and $WC[\ell] = RC[\ell]$.

Thus, the algorithm in Figure 7.2 ensures $A0$, $A1$ and $A2$, and the algorithm indeed implements an atomic register. $\square_{Theorem\ 19}$

7.4. Bibliographic notes

The construction of a multi-reader atomic register is due to Haldar and Vidyasankar [33].

7.5. Exercises

1. Show that the algorithm in Figure 7.1 does not implement an atomic register if we replace the atomic bit $WFLAG$ with a regular one.

Part III.

Snapshot objects

8. Collects and snapshots

Until now we discussed read-write abstractions in which a read operation returns the last value written to a single specified register. It would also be convenient to have an abstraction that allows the reader to get, in a single operation, the vector of the last values written by all the processes. As usual, we expect the operation to be *wait-free*, and we explore several definitions of the "last written value". We start with from the weaker *collect* object, and then proceed to the stronger *snapshot* and *immediate snapshot* objects.

8.1. Collect object

A *collect* object exports the operation $\mathit{store}()$ that is used to post values and the operation $\mathit{collect}()$ that returns a *view*, a collection of "most recent" values posted so far. More precisely, a view V is an n-vector, with one value per process. Intuitively, $\mathit{store}(v)$ is invoked by process p_i to replace the value in position i of the view with v. If no value has been posted by p_i so far, the view returned by a $\mathit{collect}()$ operation contains $\perp$ at position i.

8.1.1. Definition and implementation

A collect object can be seen as an array of n elements. Each element i can be updated by process i using the $\mathit{store}()$ operation. An evaluation of the content of the array can be obtained using the $\mathit{collect}()$ operation: each position i of the returned n-vector, called a *view*, contains the argument of a concurrent store operation or the argument of the latest store operation of p_i.

For simplicity, we assume that every value written by a given process p_i, including the initial value in position i, is unique. This way the value at position i in a view V returned by a collect operation is associated with a unique store operation s_i by p_i that has written that value, and we simply write $s_i \in V$ (the initial value $\perp$ the view is associated with an artificial "initializing" store operation performed by p_i in the beginning). We also say that view V *is contained in* a view V', and we write $V \leq V'$, if for all j, $V[j]$ is written before $V'[j]$. We write $V < V'$ if $V \leq V'$ and $V \neq V'$.

To define what does it mean for a collect object to behave correctly, consider a history H of events $\mathit{inv}[\mathit{store}()]$, $\mathit{resp}[\mathit{store}()]$, $\mathit{inv}[\mathit{collect}()]$ $\mathit{resp}[\mathit{collect}()]$ issued by the processes. Recall that $<_H$ denotes the total order on the events in H and $\rightarrow_H$ denoted the real-time order on the operations in H. As usual, we assume that H is well-formed: no process invokes a new operation on the collect object before its previous operation returns. Thus, any two operations invoked by a given process in H are related by $\rightarrow_H$. Every history H of invocations and responses on a collect object must satisfy the following properties (here C denotes a collect operation and s_i denotes a store operation of process p_i):

$B0$: For each collect operation C that returns V, and each $s_i \in V$: $C \neg \rightarrow_H s_i$. *(No collect returns a value not yet written.)*

$B1$: For each collect operation C that returns V, store operations s and s' by process p_i, such that $s' \in V$: $(s \rightarrow_H C) \Rightarrow (s = s' \ \vee \ s' \rightarrow_H s')$. *(No collect returns an overwritten value.)*

$B2$: $\forall\ V, V'$ returned by C, C': $(C \rightarrow_H C') \Rightarrow (V \leq V')$. *(Every collect contains all preceding ones.)*

A straightforward implementation of a collect object maintains n atomic registers, $REG[1], \ldots, REG[n]$, one per process. To store a value, p_i simply writes it to $REG[i]$. To collect the content, p_i reads $REG[1], \ldots, REG[n]$ in any order. We can construct a collect reading function as a composition of corresponding atomic reading functions $\pi_1, \ldots, \pi_n$: for each collect operation, define $\pi(C)[i] = \pi_i(r_i^C)$, where r_i^C is the read operation on $REG[i]$ performed within C. The reader can easily see that the resulting reading function satisfies properties $B0$–$B2$ above.

8.1.2. A collect object has no sequential specification

An abstraction A *has a sequential specification* $\mathcal{S}$, if its behavior can be expressed through a set of sequential histories in $\mathcal{S}$. Formally:

- Every implementation of A is an atomic implementation of $\mathcal{S}$, and
- Every atomic implementation of $\mathcal{S}$ is an implementation of A.

Note that the second property implies that *every* sequential history of $\mathcal{S}$ should be a history of A. If an abstraction A has a sequential implementation, we say that A is an *atomic object*.

Lemma 5 *Collect is not an atomic object.*

Proof Suppose, by contradiction, that the collect abstraction has a sequential specification $\mathcal{S}$.

Consider the execution history in Figure 8.1. Here the $collect()$ operation issued by p_1 is concurrent with two store operations issued by p_2 and p_3. The history could have been exported, for example, by an execution of the simple algorithm described above (Section 8.1.1), in which p_1, within its $collect()$ operation, reads $REG[2]$ *before* the write on $REG[2]$ performed by p_2 and $REG[3]$ *after* the write on $REG[3]$ performed by p_3.

By our assumption, the history should be atomic with respect to $\mathcal{S}$. We recall that any linearization of H should respect the real-time order on operations and, thus, we should put $[store(v)$ by $p_2]$ before $[store(v')$ by $p_3]$ in any linearization of H. We establish a contradiction by showing that there is no way to find a place for the $collect()$ operation in any such linearization.

Suppose that $\mathcal{S}$ allows placing the $collect()$ operation *before* $store(v')$ by p_3. Thus, $\mathcal{S}$ contains a sequential history that violates property $B0$ (the collect operation returns a value which is not written yet).

Now suppose that $\mathcal{S}$ allows placing the $collect()$ operation after $store(v')$ by p_3. This results in a history that violates property $B1$ (the collect operation returns an overwritten value).

In both cases, $\mathcal{S}$ contains a history that does not respect the properties of collect. $\square_{Lemma\ 5}$

Note that the proof will hold even for a weaker abstraction that only satisfies $B0$ and $B1$: a collect abstraction would not have a sequential specification even without the requirement that any collect operation should contain all preceding collect operations.

8.2. Snapshot object

One of the reasons why the collect object cannot be captured by a sequential specification is that it allows concurrent collect operations to return views that are not "ordered", i.e., not related by containment.

In this chapter, we introduce an "atomic restriction" of collect: a *snapshot* object that exports two operations: $update()$ and $snapshot()$. The $snapshot()$ operation returns a vector of n values (one per process). The value in position i of the vector contains the argument of the last preceding or a concurrent $update()$ operation executed by process p_i.

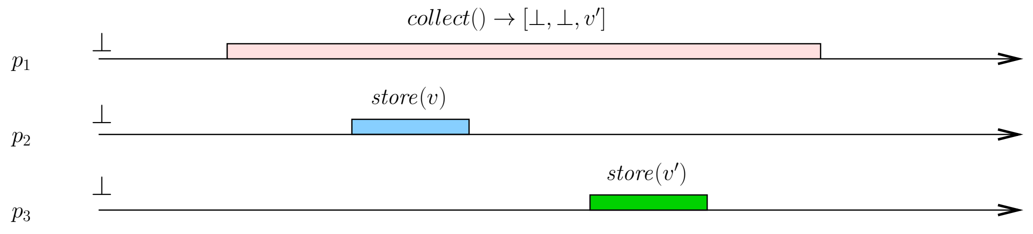


Figure 8.1.: A collect object has no sequential specification

8.2.1. Definition

In every history H, a snapshot object satisfies properties $B0$–$B2$ of collect (Section 8.1.1), where *store* and *collect* are replaced with $update$ and $snapshot$, respectively, plus the following two properties:

$B3$ For any two views V and V' obtained by snapshot operations, $(V \leq V') \vee (V' \leq V)$.

$B4$ For any two updates u and u', where u is performed by a process p_i, and any view V obtained by a snapshot operation, if $u' \in V$ and $u \rightarrow_H u'$, then V contains u or a later update at position i.

In other words, non-concurrent updates cannot be observed by snapshot operations in the opposite order: new-old inversion on the level of snapshot and updates is not allowed.

If snapshot operations S and S' return views V and V', respectively, such that $V \leq V'$, we say that S is contained in S', and write $S \leq S'$. Thus, $B3$ implies that any two snapshot operations are related by containment.

8.2.2. The sequential specification of snapshot

The sequential specification of type snapshot is defined as a set of sequential histories of $update$ and $snapshot$ operations. In every such sequential history, each position i of the vector returned by every $snapshot$ operation contains the argument of last preceding $update$ operation of p_i (if any, or the initial value $\perp$ otherwise). Note that, unlike the operational definitions of collect and snapshot objects proposed above, the definition of the sequential snapshot type is valid even if we do not assume that every value written by a given process is unique.

Intuitively, a concurrent implementation of the snapshot type gives the illusion of update and snapshot operations taking place instantaneously. We show that this type indeed captures the behavior of a snapshot object.

Lemma 6 *The snapshot abstraction is atomic (with respect to the snapshot type).*

Proof Consider a finite history H of a snapshot implementation. Recall that H satisfies properties $B0$–$B2$ of collect (where *store* and *collect* are replaced with *update* and *snapshot*), plus $B3$ and $B4$.

We construct a linearization L of H as follows. First we order all complete snapshot operations in H, based on the $\leq$ relation, which is possible by property $B3$.

Let $update(v) = U$ be an operation performed by p_i. U is then inserted in L just before the first snapshot operation that returns v or a later value in position i, or at the end of the sequence if there is no such a snapshot. After having done this for every update, we obtain a sequence $[U_0]$, S_1, $[U_1]$, S_2, $[U_2]$, …, S_k, $[U_k]$, where each $[U_j]$ is a (possibly empty) sequence of update operations U such that snapshot S_j returns values older that written by U and S_{j+1} returns the value written by U or a later value. Now we rearrange elements of each $[U_j]$ so that the real-time order is respected. This is possible since the real-time order is acyclic.

Now we show that the resulting linearization L respects the order $\rightarrow_H$. Consider two operations *op* and *op*$'$, such that *op* $\rightarrow_H$ *op*$'$. Three cases are possible:

- Both *op* and *op*$'$ are update operations. Let *op* and *op*$'$ belong to $[U_\ell]$ and $[U_m]$, respectively. If $\ell < m$, *op* $\rightarrow_L$ *op*$'$, as $[U_\ell]$ precedes $[U_m]$ in L. If $\ell = m$, L), then *op* $\rightarrow_L$ *op*$'$, as L preserves the real-time order of H in each $[U_m]$.

  Suppose now that $\ell > m$. But, by $B4$, S_{m+1} contains *op*$'$ and any update that precedes it, including *op*. By the construction of L, *op*$'$ cannot belong to U_ℓ—a contradiction.

- Both *op* and *op*$'$ are snapshot operations that return views V and V', respectively. If *op*$'$ is incomplete, then it does not appear in L. If *op*$'$ is complete, then by $B2$, $V \leq V'$. Since L orders snapshots based on the $\leq$ relation, if *op*$'$ appears in L, we have *op* $\rightarrow_L$ *op*$'$ in L.

- *op* is an update and *op*$'$ is a snapshot. By $B1$, *op*$'$ returns the value written by *op* or a later value, and, by the construction of L and $B3$, *op* $\rightarrow_L$ *op*$'$.

- *op* is a snapshot and *op*$'$ is an update. By $B0$, the value written by *op*$'$ does not appear in the result of *op*. By the construction of L, *op* $\rightarrow_L$ *op*$'$.

Thus, any snapshot object is an atomic implementation of the snapshot type.

Now consider a history H of a atomic implementation of the snapshot type. We are going to show that H satisfies properties $B0 - B4$. Let L be a linearization of H. Thus, L is a legal (with respect to the snapshot type) sequential history, that is equivalent to a completion of H and respects the real-time order in H. In particular, L contains every complete operation in H.

- Suppose that a snapshot operation S returns a value v at position i in H. Since L is legal, v is the value written by the last update u of p_i that precedes S in L. Since L respects the real-time order, S cannot precede u in H, and, thus, $B0$ is ensured in H.

- Suppose an update u precedes a snapshot S in H. Since L respects the real-time order of H, u precedes S also in L. Since L is legal, S returns the value written by u or a later value at the corresponding position and, thus, $B1$ is ensured in H.

- Suppose a snapshot S_1 precedes a snapshot S_2 in H. Since L respects the real-time order of H, S_1 precedes S_2 also in L. Legality of L implies that $S_1 \leq S_2$ and, thus, $B2$ is ensured in H.

- All complete snapshot operations appear in L and, since L is legal, are related by $\leq$: $B3$ is ensured in H.

- Suppose that an update u_1 precedes an update u_2 and a snapshot S returns the value written by u_2. Since L respects $\rightarrow_H$ and is legal, we have $u_1 \rightarrow_L u_2$ and $u_2 \rightarrow_L S$. Thus, $u_1 \rightarrow_L S$ and, since L is legal, S returns the value written by u_1 or a later value at the corresponding position: $B4$ is ensured in H.

Thus, any atomic implementation of the snapshot type is indeed a snapshot object. $\square_{Lemma\ 6}$

```
operation $update(v)$ invoked by $p_i$:
    $sn_i := sn_i + 1$            { local sequence number generator }
    $REG[i] := [v, sn_i]$          { store the pair }
```

Figure 8.2.: Update operation

```
operation $snapshot()$:
1       $aa := REG.scan()$;
2       repeat forever
3           $bb := REG.scan()$;
4           if $(aa = bb)$ then return $(aa.val)$;        { return the vector of read values }
5           $aa := bb$
```

Figure 8.3.: Snapshot operation

8.2.3. Non-blocking snapshot

We start with a simple *non-blocking* snapshot implementation that only guarantees that at least one correct process completes each of its operations. The construction assumes that the underlying base registers can store values of arbitrary (unbounded) size, i.e., we may associate ever-growing sequence numbers with every stored value. Then we turn the construction into an unbounded wait-free one. Finally, we present a wait-free snapshot implementation that uses *bounded* memory.

Our n-process snapshot implementation uses an array of atomic registers $REG[]$. Each value that can be stored in a register $REG[i]$ is associated with a sequence number that is incremented each time a new value is stored. Each $REG[i]$ consists of two fields, denoted $REG[i].sn$ and $REG[i].val$. The implementation of $update()$ is presented in Figure 8.2. Here sn_i is a local variable, initially 0, that p_i uses to generate sequence numbers.

In an update operation, process p_i simply writes the value, together with its sequence number, in the corresponding register. To ensure that the result of every snapshot operation is consistent, i.e., contains the most recent the implementation uses the "double scan" technique: the process keeps reading registers $REG[1, \dots, n]$ until two consecutive collects return identical results. The result of the last scan is then returned by the snapshot operation.

The $scan()$ function asynchronously reads the last (sequence number, data) pairs posted by each process:

```
function $REG.scan()$:
    for $j \in \{1, \dots, n\}$ do
        $R[j] := REG[j]$;
    return $(r)$
```

Theorem 20 *The algorithm in Figures 8.2 and 8.3 is a non-blocking atomic snapshot implementation.*

Proof To prove that the implementation is non-blocking, consider any infinite execution of the algorithm.

The update operation terminates in only one base-object step. Suppose now that a snapshot operation performed by a correct process p_i never terminates. By the algorithm, p_i thus executes infinitely many scans of REG. The only reason not to return in line 4 is to find out that one of the positions in REG has changed since the last scan. Thus, for every two consecutive scan operations C_1 and C_2 executed by p_i, another process p_j executes an update operation U such that write to $REG[j]$ in U takes place between the read of $REG[j]$ in C_1 and the read of $REG[j]$ in C_2. Since there are only finitely many processes, at least one process performs infinitely update operations concurrently with the snapshot operation of p_i. Thus, in every infinite execution of the algorithm, at least one correct process completes every its operation. So the implementation is indeed non-blocking.

Now we show that the implementation is linearizable with respect to the snapshot type. Let E be any finite execution of the algorithm and H be the corresponding history. Consider any complete $snapshot()$ operation in E. Let C_1 and C_2 be its last two scans. By the algorithm, C_1 and C_2 return the same result. Now we choose the linearization point of the snapshot operation to be any point in E between the response of C_1 and the invocation of C_2 (see example in Figure 8.4). Otherwise, if a snapshot operation does not return in E, we remove the operation from our completion of the corresponding history H.

Consider now an $update(v)$ operation executed by a process p_i in E. We linearize the operation at the point when it performs a write on $REG[i]$ in E (if it does not, we remove it from the completion of H).

Let L be the resulting *linearization* of H, i.e., the sequential history where operations appear in the order of their linearization points in E. By the construction, L is equivalent to a completion of H. Also, since each operation is linearized within its interval in E, L respects the real-time order of H. We show that L is legal, i.e., at every position i, every snapshot operation in L returns the value written by the latest preceding update of p_i.

Let S be a snapshot operation in L, and let C_1 and C_2 be the two last scans of S. For each p_i, let u_i be the last update operation of p_i preceding S in L. Recall that u_i is linearized at the write on $REG[i]$ and S is linearized between the response of C_1 and the invocation of C_2. Since, by the algorithm, C_1 and C_2 read the same value in $REG[i]$, no write on $REG[i]$ takes place between the read of $REG[i]$ performed within C_1 and the read of $REG[i]$ performed within C_2. Thus, since the write operation performed within u_i is the last write on $REG[i]$ to precede the linearization point of S in E, we derive that it is also the last write on $REG[i]$ to precede the read of $REG[i]$ performed within C_1.

Therefore, for each p_i, the value of p_i returned by C_1 and, thus, by S is the value written by u_i. Hence, L is legal, and the algorithm in Figures 8.2 and 8.3 gives a linearizable implementation of snapshot.

$\Box_{Theorem\ 20}$

8.2.4. Wait-free snapshot

In the non-blocking snapshot implementation in Figures 8.2 and 8.3, update operations may starve a snapshot operation out by "selfishly" updating REG. This implementation can be turned into a wait-free one using *helping*: an update operations can help concurrent snapshot operations terminate. An update operation may itself take a snapshot of and store the result together with the new value in REG. Of course, for this helping mechanism to work, we need to make sure that the intertwined snapshot and update operations do not prevent each other from terminating.

First we can make the following two observations on the non-blocking snapshot implementation:

- If two consecutive scans performed within a snapshot operation are not identical, then at least one process has concurrently performed an update operation.

- If a snapshot operation S issued by a process p_i witnesses that the value of $REG[j]$ has changed twice, i.e., p_j concurrently executed two update operations u_1 and u_2, then the second of these updates was entirely performed within the interval of S. This is because S observed the value written by u_1 (and, thus, u_2 was invoked *after* the invocation of S) and the (atomic) write by p_j of the base atomic register $REG[j]$ is the last operation of u_2.

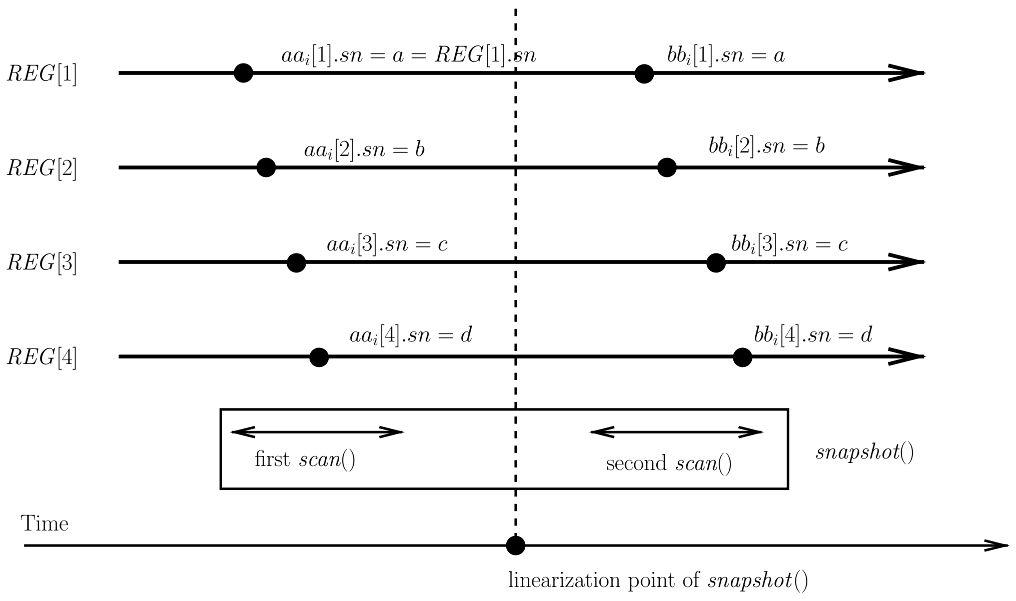


Figure 8.4.: Linearization point of a $snapshot()$ operation

As the execution interval of the second update falls entirely within the interval of S, we may use the update to “help” S as follows:

- Within u_2, p_j takes a snapshot itself (using the algorithm in Figure 8.3) and writes the result $help$ to $REG[j]$.

- Within S, p_i uses the result read in $REG[j]$ as the response of S. This is going to be a valid result, since the execution of u_2 (and, thus, of the snapshot performed by u_2) takes place entirely within the interval of S, so S can simply “borrow” the snapshot result $help$ from U_2.

Note that for this kind of helping to work, S must witness at least two concurrent updates of the same process. For example, even though the write on $REG[j]$ performed within u_1 takes place within the interval of S, the snapshot written by u_1 together with its value may have taken place way before the invocation of S. Thus, adopting the result of u_1's snapshot as the result of S may violate linearizability, since it may miss updates executed *after* the snapshot taken by u_1 but *before* the invocation of S. This is why, before adopting the snapshot taken by p_j, p_i should wait until it observes the second change in $REG[j]$.

The resulting implementations of $update()$ and $snapshot()$ are described in Figure 8.5. The atomic register $REG[i]$ consists now of three fields, $REG[i].val$ and $REG[i].sn$ as before, plus the new field $REG[i].help\_array$ that contains the result of the snapshot taken by p_i in the course of its latest update operation.

The new local variable $idcould\_help_i$ is used by process p_i when it executes $snapshot()$. Initially $\emptyset$, $idcould\_help_i$ contains the set of the processes that terminated update operations concurrently with the snapshot operation currently executed by p_i (lines 11-15). When p_i observes that a process $p_j \in could\_help$ updated its value in REG, i.e., p_i finds out that $aa_i[j].sn \neq bb_i[j].sn$, p_i returns $REG[j].help\_array$ as the result of its snapshot operation.

```
operation $update(v)$ invoked by $p_i$:
(1)  $help\_array_i := snapshot()$;
(2)  $sn_i := sn_i + 1$;
(3)  $REG[i] := (v, sn_i, help\_array_i)$

operation $snapshot()$:
(4)  $could\_help_i := \emptyset$;
(5)  $aa_i := REG.scan()$;
(6)  while $true$ do
(7)    $bb_i := REG.scan()$;
(8)    if $(\forall j \in \{1, \dots, n\} :\ aa_i[j].sn = bb_i[j].sn)$ then
(9)          return $(aa_i.val)$
(10)   else for all $j \in \{1, \dots, n\}$ do
(11)         if $(aa_i[j].sn \neq bb_i[j].sn)$ then
(12)            if $(j \in could\_help_i)$ then
(13)               return $(bb_i[j].help\_array)$
(14)            else
(15)               $could\_help_i := could\_help_i \cup \{j\}$
(16)         $aa_i := bb_i$
```

Figure 8.5.: Atomic snapshot object construction

8.2.5. The snapshot object construction is bounded wait-free

Theorem 21 *Each $update()$ or $snapshot()$ operation returns after at most $O(n^2)$ operations on base registers.*

Proof Let us first observe that an $update()$ by a correct process always terminates as long as the $snapshot()$ operation it invokes always returns. So, the proof consists in showing that any $snapshot()$ issued by a correct process p_i terminates.

Suppose, by contradiction, that a snapshot operation executed by p_i has not returned after having executed n times the **while** loop (lines 5-16). Thus, each time it has executed the loop, p_i has found out that for some new $j \notin could\_help_i$, $aa_i[j].sn \neq bb_i[j].sn$ (line 11), i.e., p_j has executed a new $update()$ operation since the last $scan()$ of p_i. After this j is added to the set $could\_help_i$ in line 14.

Note that $i \notin could\_help_i$ (p_i does not change the value of $REG[i]$ while executing $snapshot()$). Thus, after $n-1$ iterations, $could\_help_i$ contains all other $n-1$ processes $\{1, \dots, i-1, i+1, \dots, n\}$. Therefore, when p_i executes the while loop for the nth time, for any p_j such that $aa_i[j].sn \neq bb_i[j].sn$ (line 11), it finds $j \in idcould\_help_i$ in line 12. By the algorithm, p_i returns in line 13, after having executed n iterations in lines 5-16—a contradiction.

Thus, every snapshot operation returns after having executed at most n **while** loops in lines 5-16. Since every loop involves exactly n base-object reads (in the scan operation on registers $REG[1]$, ..., $REG[n]$), every snapshot terminates in n^2 base-object steps. An update operation additionally executes only one base-object write, thus its complexity is also within $O(n^2)$. $\Box_{Theorem\ 21}$

8.2.6. The snapshot object construction is atomic

Theorem 22 *The object built by the algorithms described in Figure 8.5 is atomic with respect to the* snapshot *type.*

Proof Let E be an execution of the algorithm and H be the corresponding history of E. To prove that the algorithm is indeed an atomic snapshot implementation, we construct a linearization of H, i.e., a total order L on the operations in H such that: (1) L is equivalent to a completion of H, (2) L respects the real-time order of H, and (3) L is legal, i.e., each $snapshot()$ operation S in L returns, for each process p_j, the value written by the last $update()$ operation of p_j that precedes S in L.

The desired linearization L is built as follows. The linearization point of a complete $update()$ operation in E is the write in the corresponding 1WMR register (line 3). Incomplete update operations are not included to L. The linearization point of a $snapshot()$ operation S issued by a process p_i depends on the line at which it returns.
(i) If S returns in line 9 (successful double $scan()$), then the linearization point is any time between the end of the first $scan()$ and the beginning of the second $scan()$ (see the proof of Theorem 20 and Figure 8.4).
(ii) If S returns in line 13 (i.e., p_i terminates with the help of another process p_j), then the linearization point is defined recursively as the linearization point of the corresponding update operation of p_i. In the example depicted in Figure 8.6, the arrows show the direction in which snapshot results are adopted by one operation from another.

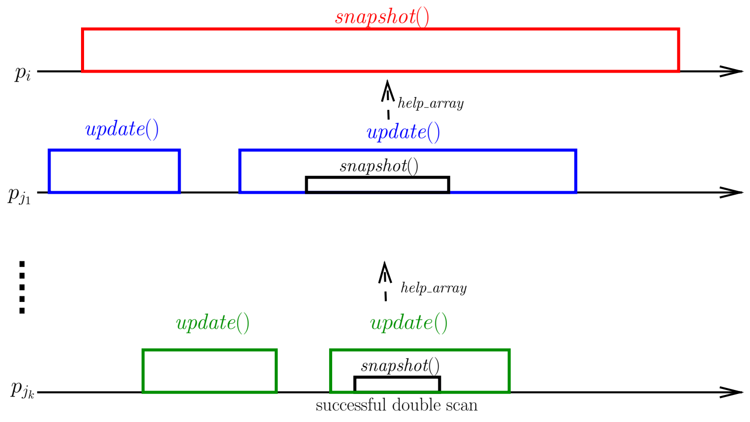


Figure 8.6.: Linearization point of a $snapshot()$ operation (case ii)

We show now that the linearization point is well-defined. If S returns in line 13, the array (say $help\_array$) returned by p_i has been provided by an $update()$ operation executed by some process p_{j_1}. As we observed earlier, this $update()$ has been entirely executed within the interval of S, since $help\_array$ is the result of the second update operation of p_j that is observed by p_i to be concurrent with S. Thus, this update started after the invocation of S and its last event (the write in $REG[j]$ in line 8) before the response of S.

Recursively, $help\_array$ has been obtained by p_{j_1} from a successful double scan, or from another process p_{j_2}. As there are at most n concurrent processes, it follows by induction that there is a process p_{j_k} that has executed a $snapshot()$ operation within the interval of S and has obtained $help\_array$ from a successful double scan.

The linearization point of the $snapshot()$ operation issued by p_i is thus defined as the linearization point of $snapshot()$ operation of p_{j_k} whose double scan determined $help\_array$.

This association of linearization points to the operations in H results in a complete sequential history L that puts the operations in H in the order their linearization points appear in E.

L trivially satisfies properties (1) and (2) stated at the beginning of the proof. Reusing the proof of Theorem 20, we observe that, for every p_j, every snapshot operation S (be it a standalone snapshot or a part of an update) returns the value written to $REG[j]$ by the last update of p_j to precede the linearization point of S in E. Thus, L also satisfies (3), and the algorithm in Figure 8.5 is an atomic implementation of snapshot. $\square_{Theorem\ 22}$

8.3. Bounded atomic snapshot

Implementing atomic abstractions is of our central concern. In Chapter 6, we described a space-optimal implementation of an atomic bit using three safe bits. In Chapter 7, we discussed how to implement a multi-valued bounded atomic registers from bounded regular registers.

In contrast, our implementation of the atomic snapshot abstraction in Section 8.2.4 assumes underlying atomic registers of *unbounded* capacity. Indeed, the values written to the abstraction by update operations are assumed to be unique, e.g., equipped with distinct sequence numbers that are taken in an unbounded range.

On can see an apparent gap between these transformations, and a natural question is whether we can use atomic registers of *bounded* size to implement atomic snapshot.

8.3.1. Double collect and helping

The unbounded construction of atomic snapshots was based on two simple ideas: *double collect* and *helping*.

Two consecutive collects returning identical results within a snapshot operation guarantee that no register has been changed in the interval of time between the return of the first collect and the invocation of the second one. Thus, all the updates affecting the result of these collects can be safely linearized before the end of the first one.

If, after taking n collects, process p_i did not observe two identical ones, then at least one of the $n-1$ other processes (let us denote it p_j) performed two concurrent updates. Now assume that each update operation of p_j includes taking a snapshot and attaching its outcome to the written snapshot value. Clearly, the snapshot attached to the second update performed by p_j and witnessed by p_i took place within the interval of the snapshot operation of p_i. Thus, it is safe for p_i to adopt this outcome as its own.

Notice, however, that these mechanisms rely on the assumption that every value written to the snapshot object is unique: otherwise two identical collects do not necessarily imply that no concurrent update took place. An amusing exercise is to find an incorrect execution of our algorithm, assuming that the "unique-write" requirement is lifted. Intuitively, the so called *ABA* problem (*A* in a snapshot position is replaced with *B* and then with *A* again, so that a concurrent reader does not see the change) may cause a snapshot operation to return an inconsistent value (see Exercise 3).

In histories with an unbounded number of updates, using a distinct value for each update operation requires unbounded memory. But suppose now that we are after a *bounded* atomic snapshot object: processes only write values from a bounded range. It turns out that a simple bounded-space *handshaking* mechanism can be used to detect modifications in a snapshot position.

8.3.2. Binary handshaking

Let us recall the signalling mechanism in the 1W1R atomic register construction (Chapter 6): the writer uses a special bit W to inform the reader that the value of the implemented register has been modified, and the reader uses another special bit R to inform the writer that the last written value has been read.

Intuitively, in an atomic snapshot construction, every process executing a snapshot operation acts as a reader, and every process executing an update operation acts as a writer. Therefore, for each distinct pair of processes (p_i, p_j), we can maintain two atomic binary registers $W[i, j]$ and $R[i, j]$, where $W[i, j]$ can be written by p_i when it performs an update and read by p_j when it performs a snapshot, while $R[i, j]$ can be written by p_j when it performs a snapshot and read by p_i when it performs an update.

Now suppose that after p_i modifies $REG[i]$, it also checks $R[i, j]$ for each $j \neq i$ and sets $W[i, j]$ to be different from $R[i, j]$. Respectively, whenever p_j collects the values of REG it checks $W[i, j]$ and, if needed, sets $R[i, j]$ to be equal to $W[i, j]$. Therefore, whenever p_j takes a subsequent scan of REG and observes $R[i, j] \neq W[i, j]$, it may deduce that $REG[i]$ has been recently changed.

It is still, however, possible that p_i changes $REG[i]$ but p_j takes its scan before p_i modifies $W[i, j]$. That is why we also introduce an additional $toggle$ bit that is attached to the value written to $REG[i]$. The bit $REG[i].toggle$ is inverted each time $REG[i]$ is written by p_i. This way p_j can detect a concurrent update operation via a change either in $REG[i].toggle$ or in $W[i, j]$.

8.3.3. Bounded snapshot using handshaking

Figure 8.7 describes a bounded implementation of the snapshot object. Now the atomic register $REG[i]$ consists of three fields, $REG[i].val$ for the written value, $REG[i].help\_array$ for the result of the snapshot taken by p_i within its latest update operation, and $REG[i].toggle$ for the bit inverted with each new update performed by p_i.

The $update$ operation is very similar to that in the unbounded algorithm (Figure 8.5). But instead of using a unique sequence number with every written value, process p_i inverts the toggle bit and makes sure that $W[i, j] \neq R[i, j]$, in order to inform every other process p_j that a new value has been written.

In the $snapshot$ operation, process p_i first ensures that $W[j, i] = R[j, i]$ for every $j \neq i$, and then performs two scans of REG. We are going to show that, for any $j \neq i$, $REG[j].toggle$ has different values in these two scans or $W[j, i]$ does not equal $R[j, i]$ if and only if $REG[j]$ has been concurrently modified. Thus, if no j satisfies the conditions in line 14, it is safe to return the outcome of the latest scan taken by p_i (line 20). If, for some j, the conditions are satisfied in *three* iterations, then it is safe to return the snapshot attached to last the value written by p_j (line 16). Note that, unlike the unbounded version (Figure 8.5), this algorithm cannot return after two concurrent modification of the shared memory performed by another process are observed (see Exercise 6).

8.3.4. Correctness

Essentially, we use the correctness arguments of the unbounded snapshot algorithm (Section 8.2.4). As before, we linearize each update operation of a process p_i at the point it writes to $REG[i]$. Each snapshot operation that detected no conflicts and returned in line 20 in any point between the end of its first scan (line 11) and the beginning of its second scan (line 12), taken just before returning. Recursively, each snapshot operation that adopts the value written by a concurrent update operation *op* (line 16) is linearized at the linearization point of the corresponding snapshot operation performed within *op* (line 1).

It remains to prove two points in this bounded algorithm though.

First, we need to show that if a snapshot operation S does not detect any change in $REG[j]$ in line 14, then indeed no $REG[j]$ has not been modified between the moment it was read in line 11 and the moment point it was read in line 12.

Lemma 7 *Let s_1 and s_2 be two consecutive scans performed within a snapshot operation S by a process p_i. If $REG[j]$ has been modified between the moment it has been read in s_1 and the moment it has been read in s_2, then the check in line 14 performed by S immediately after s_2 will succeed.*

Proof If $REG[j]$ has been modified only once after it was read in s_1 but before it was read in s_2, then the $toggle$ field is different in $aa_i[j]$ and $bb_i[j]$ and, thus, the check in line 14 will succeed.

```
operation update(v) invoked by $p_i$:
(1)  $help\_array_i := snapshot()$;
(2)  $REG[i] := (v, help\_array_i, \neg REG[i].toggle)$;
(3)  for all $j \in \{1, \ldots, n\},\ i \neq j$ do
(4)      if $R[i,j] = W[i,j]$ then
(5)          $W[i,j] := 1 - W[i,j]$

operation snapshot():
(6)  $could\_help_i := [0, \ldots, 0]$;
(7)  while $true$ do
(8)      for all $j \in \{1, \ldots, n\},\ i \neq j$ do
(9)          if $R[j,i] \neq W[j,i]$ then
(10)             $R[j,i] := 1 - R[j,i]$
(11)     $aa_i := REG.scan()$;
(12)     $bb_i := REG.scan()$;
(13)     for all $j \in \{1, \ldots, n\},\ i \neq j$ do
(14)         if $R[j,i] \neq W[j,i]$ or
                 $aa_i[j].toggle \neq bb_i[j].toggle$ then
(15)             if $could\_help_i[j] = 2$ then
(16)                 return $(REG[j].help\_array)$
(17)             else
(18)                 $could\_help_i[j] := could\_help_i[j] + 1$
(19)         else
(20)             return $(bb_i.val)$
```

Figure 8.7.: Bounded atomic snapshot

Suppose now that $REG[j]$ has been modified twice or more in the chosen interval. By the update algorithm, between any two modifications of $REG[j]$, p_j must make sure that $R[j,i] \neq W[j,i]$ (lines 3-5). Since between s_1 and s_2, p_i does not modify $R[j,i]$, when it reads $W[j,i]$ immediately after the scans (line 14), it will find $R[j,i] \neq W[j,i]$ in line 14 and the check will succeed. $\Box_{Lemma\ 7}$

Thus, a snapshot operation that, for all j, passed through the checks in line 14 and returned in line 20 can be safely linearized at any point between its last two scans.

Second, we need to show that it is also safe to a snapshot operation to "borrow" the outcome of a snapshot taken by a process that has been witnessed "moving" three times (line 16). within the interval of S. For this, we first prove the following auxiliary result:

Lemma 8 *Let s_1 and s_2 be two consecutive scans performed within a snapshot operation S by a process p_i (lines 11 and 12). If the check in line 14 performed by S immediately after s_2 succeeds for some j, then $REG[j]$ or $W[j,i]$ has been modified in the interval between time t_1, when $W[j,i]$ has been read just by p_i before s_1 (line 9), and time t_2, when $W[j,i]$ has been read by p_i just after s_2 (line 14).*

Proof Suppose that the check in line 14 succeeds because the toggle bit of $REG[j]$ has changed. This can only happen if p_j has written to $REG[j]$ (line 2)) between the reads of the register performed by p_i within s_1 and s_2 and, thus, in the desired interval.

Suppose now that p_i finds out, in line 14, that $R[j,i] \neq W[j,i]$. But after having read $W[j,i]$ at time t_1 and before executing s_1, p_i has made sure that $R[j,i] = W[j,i]$ (lines 9 and 10. Thus, the only reason to find out later that $R[j,i] \neq W[j,i]$ can be a modification of $W[j,i]$ (line 5) performed in the interval between t_1 and t_2. $\Box_{Lemma\ 8}$

Lemma 9 *If a snapshot operation S returns the view provided by an update operation U (line 16), then the execution of the snapshot S' taken by U falls within the interval of S.*

Proof Suppose that p_i, within a snapshot operation S, returns the view written by an update operation U performed by p_j. By the algorithm and Lemma 8, during S, p_j "moved" (by modifying $REG[j]$ or $W[j, i]$) at least three times.

Note that p_j can modify each of the registers $REG[j]$ and $W[j, i]$ at most once during an update operation: in lines 2 and 5, respectively. Thus, if three checks in line 14 performed by S succeed, the *first* and the *third* modifications of $REG[j]$ and $W[j, i]$ witnessed by S must belong to different update operations performed by p_j, let us denote these update operations by U_1 and U_2.

Since an update operation performed by p_j first takes a snapshot, then writes the outcome to $REG[j]$ (together with its value and the toggle bit), and then modifies $W[j, i]$ (if needed), we conclude that the value read by S in $REG[j]$ in line 16 was written by a concurrent operation U, which is U_2 or a subsequent update operation. But since U_1 is concurrent with S and U succeeds U_1, we have that the snapshot operation S' taken within U is entirely contained within the interval of S. $\square_{Lemma\ 9}$

Thus, we can safely assign the linearization point of S to the linearization point of S'. As in the unbounded case, this recursive assignment of linearization points to snapshot operations is well-defined. The reader is encouraged to check this and to show that the sequential history based on these linearization points is legal, following the proof for the unbounded algorithm.

8.4. Bibliographic notes

The collect abstraction was introduced by Aspnes and Waarts [5], refined and implemented in an adaptive way by Attiya, Fouren, and Gafni [7]. The notion of atomic snapshot was introduced by Afek et al. in [1].

Exercises

1. Would the algorithm implementing collect (Section 8.1.1) be correct if instead of atomic registers regular ones were used?

   If not, would it be correct if we only require properties $B0$ and $B1$ to be satisfied?

2. Give a *sequentially consistent* wait-free implementation of atomic snapshot with $O(n)$ step complexity.

3. Show that the non-blocking atomic snapshot algorithm (Section 8.2.3) is not correct if the values of update operations are not unique.

   Hint: consider an instance of the classical *ABA problem*: a register is written with value A, then overwritten with value B, and then overwritten with A again, so that a concurrent reader reading A and then A again cannot detect that the register temporarily stored B.

4. Show that the bounded implementation of atomic snapshot (Section 8.3) is not correct if we do not use *toggle* bits.

5. Show, by presenting a counter-example, that the bounded snapshot algorithm (Figure 8.7) would be incorrect if we did not use the toggle bit.

6. Show that the bounded algorithm is incorrect if the condition in line 15 is replaced with $could\_help_i[j] = 1$.

7. Show that the bounded algorithm is incorrect if line 16 is replaced with *return* $(bb_i[j].help\_array)$.

9. Immediate snapshot and iterated immediate snapshot

In Chapter 8, we discussed the atomic-snapshot abstraction that provides two operations, *update*, which allows a process to write a value in a dedicated memory location, and *snapshot*, which atomically returns the "current" state of the memory. Strong and useful, the atomic-snapshot abstraction, however, does not preclude a situation when snapshots taken by different processes are "unbalanced": a snapshot S_i taken by p_i contains a value written by p_j but the snapshot S_j taken by p_j contains more recent values (and, thus, is more up-to-date) than S_i. In this chapter, we discuss a restricted version of atomic snapshot, called *immediate snapshot*, that only exports "balanced" runs: IF p_i "sees" p_j, than S_i contains S_j.

9.1. Immediate snapshots

9.1.1. Definition

An immediate-snapshot object exports a single operation *update_snapshot*$()$ that takes a value as a parameter and returns a vector of values (a *view*) in response. It is required that the executions of these operations appear as executed in "batches". In each batch, a fixed subset of processes execute their *update_snapshot*$()$ *in parallel*: the processes in the subset first execute their updates and then take their snapshots. Obviously, the results of the snapshots taken by the processes in the same batch are identical. Intuitively, these snapshots are operations "immediate" in the sense that the snapshot taken by a process does not "lag" too much behind its update. As we shall see, the immediate-snapshot model has a straightforward geometrical representation which, in turn, enables simple and elegant reasoning about the model's computability.

As in the original definition of atomic snapshots (Chapter 8), we assume that each written value is unique. Any history of an immediate-snapshot object satisfies the following properties.

- **Self-inclusion.** For any operation *update_snapshot*(v_i) that returns V_i, we have $(i, v_i) \in V_i$.
- **Containment.** For any two operations *update_snapshot*(v_i) and *update_snapshot*(v_j) that return V_i and V_j, respectively, we have $V_i \leq V_j$ or $V_j \leq V_i$.
- **Immediacy.** For any operation *update_snapshot*(v_i) and *update_snapshot*(v_j) that return V_i and V_j, respectively, if $(i, v_i) \in V_j$ then $V_j \leq V_i$.

The first two properties will automatically hold if we take an atomic snapshot object and implement *update_snapshot*(v_i) as $update(v_i)$ followed by $snapshot()$. However, the immediacy property will not be satisfied here: it is possible that an update operation of a process p_i is followed by an update and snapshot operation of another process p_j, and then multiple updates and snapshots of other processes (see, e.g., Figure 9.1). The subsequent snapshot by p_i would then strictly succeed the snapshot taken by p_j, as it would contain the updates that occurred after p_j performed its snapshot (see Exercise 3).

Notice that the immediacy property implies that the immediate snapshot object has no sequential specification. Indeed, a history in which *update_snapshot*(v_i) and *update_snapshot*(v_j) return V_i and V_j, respectively, such that $(i, v_i) \in V_j$ and $(j, v_j) \in V_j$ does not allow for a legal ordering of these two

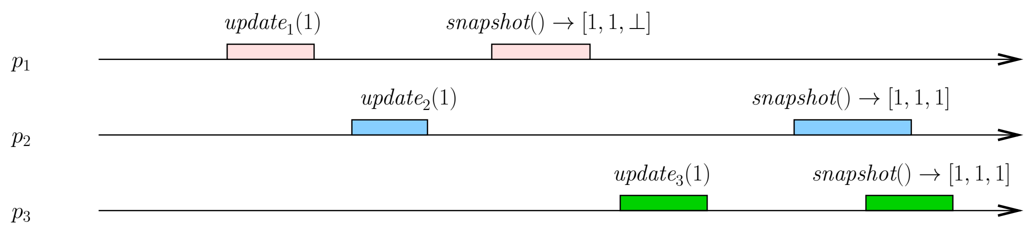


Figure 9.1.: An example of an "unbalanced" execution: p_1 sees p_2 but misses p_2's snapshot

operations with a sequential semantics that matches the properties above. We leave it to the reader to prove this claim, e.g., along the lines of the proof of Lemma 5 (Exercise 1).

9.1.2. Block runs

We can view the immediate-snapshot model as a *subset* of runs of the conventional atomic-snapshot model in which every process alternates between performing updates (on its distinct location in the shared memory) and taking atomic snapshots. Every run in the immediate-snapshot model is induced by a *block sequence*:

$$B_1, B_2, B_2, \dots,$$

where each B_i is a non-empty set of processes. The induced run consists in B_1 performing updates (in an arbitrary order) and then taking snapshots (in the arbitrary order), followed by all processes in B_2 performing updates and then taking snapshots, and so on.

It is not hard to see that the snapshots taken by the members of the same B_i are identical and for all $i < j$, the snapshot V_i taken by B_i and the snapshot V_j taken by B_j satisfy $V_i \leq V_j$. Moreover, if V_i only contains values that processes in B_j, $j \leq i$ have written in the induced run. Thus, if $(i, v_i) \in V_j$, where v_i is the value written by p_i just before it obtained immediate snapshot V_i, then $V_i \leq V_j$.

9.1.3. A one-shot implementation

We begin with an implementation of the immediate-snapshot abstraction, assuming that every process performs at most one *update_snapshot*() in a run.

The algorithm, presented in Figure 9.2, uses a shared array of 1WMR atomic registers $REG[1:n]$, where $REFG[i]$ can be written only by p_i and read by all processes. Each $REFG[i]$ stores a pair (ℓ_i, v_i), initially $(n+1, \perp)$, where v_i is the value written by p_i and ℓ_i is the *level* reached by p_i so far.

Operation

The algorithm operates as follows. Every process p_i begins with *posting* its value v_i in $VAL[i]$ and announcing its participating at level n by writing n in $REG[i]$ and . Then it reads $REG[1:n]$ to check the levels reached by other processes. If all n processes are at levels n or less, then p_i returns the set of n their values (read in VAL). Otherwise, p_i goes down to level $n-1$. If, inductively, after writing ℓ ($\ell = n-1, \dots, 1$) in $REG[i]$ and checking $REG[1:n]$, p_i finds out that ℓ processes reached levels ℓ or lower, it returns the values of these ℓ processes. Clearly, the process returns at level 1 at the latest, i.e., the algorithm is *bounded* wait-free: it takes $O(n^2)$ basic reads and writes to complete an operation.

```
Shared:
        value array of registers $VAL[1,\dots,n]$, initially $\perp$
        integer array of registers $REG[1,\dots,n]$, initially $n+1$

Local:
        value array $val[1,\dots,n]$, initially $\perp$
        integer $level$, initially $n+1$

operation $update\_snapshot(v_i)$ invoked by $p_i$:
        $VAL[i] := v_i$
(1)     repeat
(2)             $level := level - 1$
(3)             $REG[i] := level$
(4)             $V := \emptyset$
(5)             for_each $j \in \{1,\dots,n\}$ do
                        $\ell := REG[j]$
                        if $\ell \leq level$ then $V := V \cup \{j\}$
(6)     until $|V| \geq level$
(7)     for_each $j \in \{1,\dots,n\}$ do
                if $j \in V$ then $val[j] := VAL[j]$
(8)     return $(val)$
```

Figure 9.2.: A one-shot IS implementation

Correctness

To get an intuition about the algorithm's correctness, let us consider a run in which a set of k processes proceed in *lock step*, i.e., the k processes alternate between concurrently writing to REG and reading $REG[1:n]$. Notice that in this run, whenever a process reaches a level ℓ and reads $REG[1:n]$, it witnesses exactly k processes at the same level. Thus, all the processes will return the same set of k values as soon as they reach level k.

At the other extreme, consider a sequential execution of n processes performing *update_snapshot*() operations one by one. The first process, as it only sees itself, will be obliged to return at level 1. Inductively, the k-th process in the sequential order ($k = 2,\dots,n$), will output at level k: it will see itself and $k-1$ processes before it. Thus, the processes will return strictly increasing sets of values, from a singleton containing the value of the first process to the

More generally, the last process p_i to *reach* level n, i.e., to write (n, v_i) in $REG[i]$ will see exactly n processes at levels n or lower. Thus, p_i returns the set of n values, and at most $n-1$ processes will reach levels $n-1$ or lower. Inductively, we can show that if ℓ processes reach level ℓ ($\ell = n,\dots,2$), at least one process will return at this level, and at most $\ell-1$ will proceed to level $\ell-1$.

Formally, what we need to show is that, in every run of the algorithm, the sets of values returned by the processes satisfy the three properties of immediate snapshot: self-inclusion, containment and immediacy.

Lemma 10 *The algorithm in Figure 9.2 is bounded wait-free.*

Proof In every round (lines 1–6), a process performs one write and n reads. In the round n (reaching at level 1), the process will see at least one value (its own). Thus, at the latest, the process returns in round n and, thus, every operation performs $O(n^2)$ basic read-write steps. $\square_{Lemma\ 10}$

Consider any run of the algorithm. Let S_ℓ denote the set of processes that ever reach level ℓ in that run. By the algorithm, $S_1 \subseteq S_2 \subseteq \dots \subseteq S_n$.

Lemma 11 *For all $\ell \in \{1, \ldots, n\}$, $|S_\ell| \leq \ell$.*

Proof We proceed by downward induction on ℓ. The base case $\ell = n$ is trivial, as there are at most n processes taking steps in any run.

Suppose that for some $\ell \in \{2, \ldots, n\}$, $|S_\ell| \leq \ell$, i.e., at most ℓ processes reach level ℓ. If $|S_\ell| < \ell$, then we are done, as $S_{\ell-1} \subseteq S_\ell$. Otherwise, suppose that $|S_\ell| = \ell$, and let p_j be the last process in this set of ℓ processes that reaches level ℓ, i.e., writes ℓ in REG in line 3. By the algorithm, p_j witnesses exactly n processes at levels ℓ and lower and, thus, returns in level ℓ. Therefore, at most $\ell - 1$ process ever reach level $\ell - 1$. $\square_{Lemma\ 11}$

Theorem 23 *The algorithm in Figure 9.2 is a bounded wait-free implementation of immediate snapshot.*

Proof By Lemma 10, the algorithm is bounded wait-free.

Consider any run of the algorithm, and let V_i denote the set of values returned by a process p_i in that run. Let ℓ_i denote the level at which p_i returns. By the algorithm, p_i reached level ℓ_i by writing ℓ_i in $REG[i]$, then read $REG[1:n]$ and then returned the set of ℓ_i values written by processes that reached level ℓ_i or lower.

Thus, p_i returned values written by a subset of S_{ℓ_i} of size ℓ_i or more, including its own value—the property of **self-inclusion** is ensured. Furthermore, by Lemma 11, $S_{\ell_i} \leq \ell_i$ and, thus, p_i returned *exactly* the values of processes in S_{ℓ_i}.

Consider any other process p_j that returned in the given run and suppose, without loss of generality, that p_j returned at level $\ell_j < \ell_i$. Recall that $S_{\ell_j} \subseteq S_{\ell_i}$ and, thus, $V_j \subseteq V_i$—the property of **containment** is ensured.

Finally, consider any process p_j such that $p_j \in S_{\ell_i}$ and, thus, $v_j \in V_i$. Since p_j reached level ℓ_i in that run, it can only return some the values written by some S_{ℓ_j} such that $\ell_j \leq \ell_i$. Since , $S_{\ell_j} \subseteq S_{\ell_i}$, we have $V_j \subseteq V_i$—the property of **immediacy** is ensured. $\square_{Theorem\ 23}$

9.2. Fast renaming

To illustrate how the IS model can be used, we describe an elegant algorithm solving the classical *renaming* problem. In the renaming algorithm, processes take, as inputs, with *original names* from a large range and return, as outputs, *new names* taken in a smaller range the size of which is proportional to the number of participating processes. More precisely, the following properties must be satisfied in every run of a renaming algorithm:

- *Termination:* Every correct process eventually output a name.
- *Uniqueness:* Now two distinct processes output the same name.
- *Name-Adaptivity:* The output names belong to the range $\{1, \ldots, 2p-1\}$, where p is the number of participating processes.

To rule out a trivial solution in which process p_i outputs name i we add the following requirement:

- *Anonymity:* For all p_i and p_j, the algorithm of p_i with input x is the same as the algorithm of p_j with input x.

We should be careful here. In solving renaming, assuming that a single-writer multi-reader share memory is available somewhat undermines the very motivation behind this problem that, even though there is a bound on the number of participating processes in every run, the participants themselves may come from a very large (unbounded) space. One may ask how the assignment of distinct single-writer registers to participating can be implemented in such a system. The challenge of simulating single-writer multi-reader memory in such a system (also called *bootstraping*) has been addressed in [20, 21]. In this chapter, we however rule this out by assuming anonymous algorithms.

9.2.1. Snapshot-based renaming

A simple snapshot-based renaming algorithm in Figure 9.3 is based on "arbitration". A process starts with writing its input name in its dedicated register. Then it takes a snapshot of the memory to evaluate the set of participants, selects a name based on its *ranking* in the set (using the *compare* operator), writes the chosen name, together with its input, back in its register, and takes a snapshot again. If no other process chose the same name, the process terminates with the chosen output. Otherwise, the process chooses, as its new name, the first name with its ranking in the current set of participants that is not *claimed* by another process and repeats the procedure.

```
Shared:
    atomic-snapshot object $AS$


operation $rename(v_i)$ invoked by $p_i$ with input $v_i$:
    $name := 1$
    repeat forever
        $AS.update([name, v_i])$
        $S := AS.snapshot()$
        if $S$ contains no $[name', v_j]$ such that $name' = name$ and $v_j \neq v_i$ then
            return name
        $rank :=$ the rank of $v_i$ in $\{v_j \mid [*, v_j] \in S\}$
        $free := \{u \mid [u, *] \notin S\}$
        $name :=$ the $r$-th element in $free$
```

Figure 9.3.: A renaming algorithm using atomic snapshots

When p processes participating, the largest name a process may choose is $2p-1$. Intuitively, a given process can "block" at most two names at a time: one it has written to the memory and one that it is about to write. As a result, in the worst case, the process may see $p-1$ blocked and have rank p among the participants: thus, the largest name $2p-1$.

9.2.2. IS-based renaming

In the recursive IS-based algorithm described in Figure 9.3, we use one-shot IS instances to evaluate the set of participating processes. Each invocation of the IS instance is associated with a range of names that the processes invoking this instance are allowed to return. The range is determined via a starting point (denoted *start*) and a *direction* (denoted *dir* $\in \{-1, 1\}$) in which names of the range, starting from *start*, are allocated. A list of integer values *tags* contains the sequence of starting points of preceding recursive calls of *get_name*.

For instance, if p processes invoke *get_name*$(tags, start, dir)$, then the algorithm guarantees that all names output by these processes fall within the range $start + dir, \ldots, start + dir(2p - 1)$ of $2p - 1$ names.

The property of IS that the number of processes that output a set of values of size ℓ is precisely ℓ minus the number of processes that output strictly smaller sets of values guarantees that all output names are distinct.

For each sequence L of values in $\{1, \ldots, n\}$, the algorithm uses a distinct one-shot IS object $IS[L]$. A process invokes *get_name*(L, f, d) where L is the list of sizes of sets obtained in all preceding IS calls. As we will show, all such sequences L are monotonically decreasing.

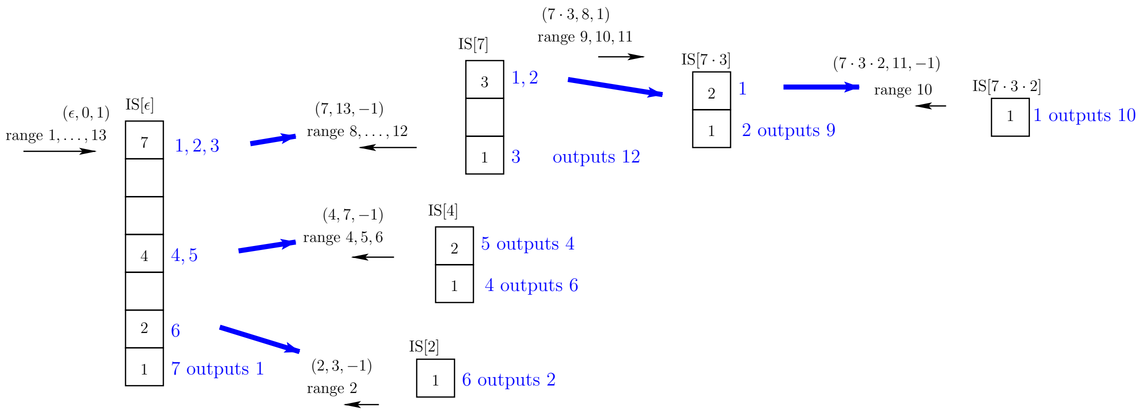


Figure 9.4.: An execution of the renaming algorithm in Figure 9.5

The get a new name, every process p_i invokes *get_name*$(\epsilon, 0, 1)$, where ϵ is the empty list. Within *get_name*$(L, start, dir)$, the process first invokes $IS[L]$.*update_snapshot*(v_i), where v_i is its input name, to get a set S of input names. If v_i happens to be the largest name in S, p_i returns the "most far-away" name in the range $start + dir, \ldots, start + dir(2|S| - 1)$, i.e., $name = start + dir(2|S| - 1)$. Otherwise, p_i selects *name* as a new starting point and inverses the direction by recursively calling *get_name*$(L \cdot |S|, name, -dir)$ to get its new name.

In Figure 9.4, we describe an execution of the algorithm for seven processes with original names $1, \ldots, 7$. The processes invoke *get_name* with parameters $(\epsilon, 0, 1)$ which means that they compete for names in the range $1, \ldots, 13\}$. Suppose that after accessing $IS[\epsilon]$, processes with names 1, 2 and 3 see all 7 processes, processes with names 4, 5 see four processes 4, 5, 6, 7, processwith name 6 sees 6, 7 and process with name 7 sees only itself.

As their names are not the maximal in the set, 1, 2 and 3 invoke *get_name* with parameters $(7, 13, -1)$, i.e., they compete for names in the range 12, 11, 10, 9, 8 (in the descending order). After accessing $IS[7]$, process with name 3 sees only itself and outputs 12 (the "first" name in the range). Processes with names 1 and 2 see all of the three processes and invoke *get_name* with parameters $(7 \cdot 3, 8, 1)$ to compete for names in the range 9, 10, 11 (in the ascending order).

After accessing $IS[7 \cdot 3]$, process 2 sees only itself and outputs 9 (the "first" name in the corresponding range). Process with name 1 sees both 1 and 2 and, thus, invokes *get_name*$(7 \cdot 3 \cdot 2, 11, -1)$ to finally output 10.

Given that an access to one-shot IS object exhibits $O(n^2)$ read-write steps, we get the following result.

Lemma 12 *In every run of the renaming algorithm in Figure 9.5, every correct process returns in $O(n^2)$ read-write steps.*

```
Shared:
        for each $L$, list of values in $\{1,\dots,n\}$: one-shot IS instance $IS[L]$

operation $get\_name(L, start, \mathrm{dir})$ invoked by $p_i$ with input $v_i$:
 (1)    $S := IS[L].update\_snapshot(v_i)$
 (2)    $st := start + dir(2|S|-1)$
 (3)    if $v_i = \max(S)$ then
 (4)            $name := st$
        else
 (5)            $name := get\_name(L\cdot|S|, st, -dir)$
 (6)    return $name$

operation $rename(v_i)$ invoked by $p_i$ with input $v_i$:
 (7)    return $get\_name(\epsilon, 0, 1)$
```

Figure 9.5.: A renaming algorithm using one-shot IS instances

Proof By the algorithm, the participating processes start with calling *get_name*$(\epsilon, 0, 1)$. We observe first that the participant with the highest input name will return the value computed in line 6 of this call. Indeed, regardless of the set of participating processes, it obtains in line 1, it will always itself to have the maximal name. The property holds for any recursive call of *get_name* (line 5). Thus, the number of processes that reach line 5 within a call of *get_name* is at least by one smaller than the number of processes that started this call. When the total number of processes preforming a call of *get_name*$(L, start, \mathrm{dir})$ drops to one, this process will return the value computed in 6.

Thus, in the worst case, a process returns in the n-th recursive calls of *get_name*. Each recursive call involves a single invocation of a single invocation of *update_snapshot* on a one-shot IS instance which gives $O(n^2)$ read-write complexity per instance and, thus, $O(n^3)$ total step complexity per call of *rename*(v_i). $\square_{Lemma\ 12}$

The safety properties of renaming (*Uniqueness* and *Name-Adaptivity*) are shown via the following auxiliary lemma:

Lemma 13 *Suppose that at most $k > 0$ processes call get_name(L, s, d) in a run of the algorithm in Figure 9.5. Then these calls can only return distinct values outside $\{s+d, \dots, s+d(2k-1)\}$.*

Proof We note first that, since the size of the set returned by a one-shot IS instance unambiguously identifies the set itself, every two processes that call *get_name*$(L, -, -)$ agree on the remaining two parameters.

Now we proceed by induction on k. The claim holds trivially when $k = 1$: the only process to call *get_name*$(L, start, \mathrm{dir})$ obtains a set of size 1 from *IS*$[L]$ and returns value *start* + *dir* computed in line 6.

Now suppose that the claim holds for all values $k' < k$ and consider a run in which k processes call *get_name*$(L, start, \mathrm{dir})$.

Suppose that the k processes obtained sets of distinct sizes $1 \leq \ell_1 < \dots < \ell_m$ from *IS*$[L]$.

We can show that $\ell_m = k$ and if $m \leq 2$, then for all $j = 2\dots, m$, the number of processes that obtained a set of size ℓ_j is $\ell_j - \ell_{j-1}$. We leave it to the reader to prove this claim (Exercise 2).

Note that the process with the highest input name that obtained the set of size ℓ_1 will output a value. Thus, at most $\ell_1 - 1 < k$ processes can recursively call *get_name*$(L \cdot \ell_1, s + d(2\ell_1 - 1), -d)$. If $\ell_1 > 1$, by the induction hypothesis, these at most $\ell_1 - 1$ processes can only get names in the range $\{s + d(2\ell_1 - 1) - d, \dots, s + d(2\ell_1 - 1) - d(2(\ell_1 - 1) - 1)\} = \{s + d, \dots, s + d(2\ell_1 - 2)\} \subseteq \{s + d, \dots, s + d(2k - 1)\}$.

Now suppose that $m \geq 2$ and consider $j = 2, \ldots, m$. By the algorithm, at most $\ell_j - \ell_{j-1} < k$ can recursively call *get_name*$(L \cdot \ell_j, s + d(2\ell_j - 1), -d)$ which, by the induction hypothesis, can only return names in the range $\{s + d(2\ell_j - 1) - d, \ldots, s + d(2\ell_j - 1) - d(2(\ell_j - \ell_{j-1}) - 1)\} = \{s + 2\ell_{j-1}d, \ldots, s + d(2\ell_j - 2)\}$ which, as $1 \leq \ell_{j-1} < \ell_j \leq k$, is a subset of $\{s + d, \ldots, s + d(2k - 1)\}$.

Thus, all outputs of recursive calls of *get_name* are distinct subsets of non-overlapping ranges $\{s + d, \ldots, s + 2\ell_1 d - 2d)\}$, $\{s + 2\ell_1 d, \ldots, s + 2\ell_2 d - 2d\}, \ldots$, $\{s + 2\ell_{m-1}d, \ldots, s + 2\ell_m d - 2d\}$, all of which are subsets of $\{s + d, \ldots, s + d(2k - 1)\}$. Moreover, the output names computed in line 4 belong to the set $\{s + d(2\ell_1 - 1), \ldots, s + d(2\ell_m - 1)\}$ which does not intersect with the ranges above. Hence, all outputs values are distinct and belong to $\{s + d, \ldots, s + d(2k - 1)\}$. $\Box_{Lemma\ 13}$

We are finally ready to prove that our algorithm is correct.

Theorem 24 *The algorithm in Figure 9.5 solves renaming with $O(n^3)$ read-write step complexity.*

Proof Consider any run of the algorithm. By Lemma 12, every correct process returns in $O(n^3)$ steps—the *Termination* property holds.

Suppose that p processes participate. Since every process obtains a new name by calling *get_name*$(\epsilon, 0, 1)$, Lemma 13 implies that all output names are distinct and belong to $\{1, \ldots, 2p-1\}$—the *Uniqueness* and *Name-Adaptivity* properties are satisfied. Finally, the algorithm only uses input names and not process identifiers, ensuring the *Anonymity* property. $\Box_{Theorem\ 24}$

9.3. Long-lived immediate snapshot

The immediate-snapshot (IS) model is at least as powerful as the classical read-write one. Assuming the full-information protocol (every written value contains the outcome of the most recent *update_snapshot*() operation), a run the IS model can be represented as a run of the full-information Atomic-Snapshot model. Thus, anything that can be solved in the AS model, can also be solved in the IS one.

In this section, we show that the inverse is also true. We present an algorithm that, in the AS model, simulates a run of IS model.

9.3.1. Overview of the algorithm

The idea behind our simulation is to use the one-shot implementation in Figure 9.2 on an *unbounded* number of *floors*. Intuitively, each floor corresponds to the total number of write operations a process completed at a given point of a run. For simplicity, we assume that every process maintains a local counter (initially 0) that is incremented and used as an argument each time the *update_snapshot* operation is invoked. The operation returns a *view*: an array of counter values of all the processes.

In the *update_snapshot* operation, every process p_i first updates a snapshot memory with its current counter valus, takes a snapshot V. The starting floor s for p_i is then computed as the sum of counter values in V: $\sum_j V[j]$. The process then *registers* its view at floor s (line 4) and, starting from floor $s - 1$ downwards, accesses IS instances until it finds a registered view with a previous value of p_i that was "seen" by some process in the view obtained from the IS instance at that floor. At this moment, p_i returns a view constructed as a "maximum" of the registered view at that floor and the view returned by the corresponding one-shot IS instance.

More precisely, for each floor f we maintained the following shared variables:

- $view_f$, used to register the view associated with this floor;
- IS_f, a one-shot IS instance;

- $\mathit{flag}_f[1, \ldots, n]$, an array of boolean *flags*, one for each process. The flag is used to signal that a non-$\perp$ value is written in view_f by a concurrent process.

When a process p_i enters a floor f, it first check if there is a registered view ($\mathit{view}_f \neq \perp$) and stores the result in $\mathit{flag}_f[i]$. Then it gets a view W of concurrently active processes in IS_f. To return at floor f, p_i must ensure that at least one process in W has witnessed a previous value of p_i in view_f.

To ensure that the first invocation of *update_snapshot* of every process returns at floor 0 or higher, we initialize view_0 with $[0, \ldots, 0]$ and flag_0 with $[\mathit{true}, \ldots, \mathit{true}]$. For $f > 1$, view_f initially stores $\perp$ and flag_f initially stores $[\mathit{false}, \ldots, \mathit{false}]$.

```
**Shared**:
        $C$, a collect object, each position $C[i]$ is a counter value for $p_i$
        For each floor $f \in \mathbb{N}$:
                $IS_f$, one-shot IS instance
                $view_f$, register storing a view, initially $\perp$ for $f > 0$ and $[0, \ldots, 0]$ for $f = 0$
                $flag_f[1, \ldots, n]$, array of boolean registers, initially $[false, \ldots, false]$ for $f > 0$ and $[true, \ldots, true]$ for $f = 0$


**operation** $update\_snapshot(count)$ **invoked by** $p_i$:
(1)     $C.update(count)$            { publish a new distinct value }
(2)     $V := C.snapshot()$            { get a view }
(3)     $f := \sum_j V[j]$            { compute the starting floor }
(4)     $view_f := V$            { register at floor $f$ }
(5)     $flag_f[i] := true$            { set the flag at floor $f$ }
(6)     **repeat forever**
(7)             $f := f - 1$
(8)             $flag_f[i] := (view[f] \neq \perp)$            { check if any process started at level $f$ }
(9)             $W := IS_f.update\_snapshot(count)$            { Access IS at floor $f$ }
(10)            **if** (for some $j \in W, flag_f[j] = true$ **and** $count > view_f[i]$) **then**
(11)                    return $\max(W, view_f)$            { take the maximum of the two views }
```

Figure 9.6.: A long-lived IS memory implementation

9.3.2. Proof of correctness

First we observe that views registered different floors are related by containment.

Lemma 14 *Let V and V' be views written, respectively, in view_f and $\mathit{view}_{f'}$, such that $f \leq f'$. Then $V \leq V'$. Also, if $f < f'$, then $V' \not\leq V$.*

Proof Recall that every value V written in variables view_f is a snapshot of atomic-snapshot memory C of size f, i.e., $\sum_j V[j] = f$. Since all such snapshots are related by containment, *every* value written in view_f can only be V, and every value V' witten in $\mathit{view}_{f'}$ $f' > f$ must satisfy $V \leq V'$. $\square_{Lemma\ 14}$

We should first that the algorithm uses the one-shot IS instances correctly, i.e., no process accesses a given instance more than once.

Lemma 15 *No one-shot instance IS_f is invoked more than once by a given process.*

Proof A given invocation of *update_snapshot* by a process p_i involves at most one invocation of IS_f (when it reaches floor f).

What remains to show is that different invocations of *update_snapshot* by p_i, op_1 and op_2, do not invoke the same IS instance. Suppose that op_1 starts at floor f (line 3). By the algorithm, within op_1, p_i writes a view containing its value in $view_f$ and accesses IS at floors $< f$.

A subsequent operation op_2 will take a snapshot of written values containing a strictly higher value for p_i and, thus, will start at floor $> f$. Within op_2, p_i either returns at a floor $> f$ or reaches floor f and, by Lemma 14, finds out that $view_f$ contains its previous value and returns before accessing lower floors. $\Box_{Lemma\ 15}$

Lemma 16 *The algorithm in Figure 9.6 satisfies the Self-inclusion property.*

Proof Let p_i return from an invocation of *update_snapshot*$(count)$ at floor f. The returned view V contains the maximun between the values for p_i found in $view_f$ and the view W returned by IS_f (line 11).

By the condition in line 10, $view_f$ contains an value $< count$. By the Self-inclusion property of one-shot IS instances, $(i, count) \in W$ and, thus, $(i, count) \in V$. $\Box_{Lemma\ 16}$

Lemma 17 *Let p_i return V_i at floor f_i and p_j return V_j at floor f_j, such that $f_i < f_j$. Then $V_i \leq V_j$.*

Proof By Lemma 14, $view_{f_i} \leq view_{p_j}$.

Let W_i be the value obtained by p_i from IS_{f_i}. We are going to show that $W_i[k] \leq V_j[k]$, for all $p_k \in W_i$, i.e., every value found in W_k is at most as recent as $V_j[k]$. Suppose that $W_i[k] = (k, v)$, i.e., p_k invoked IS_{f_i} with the argument of its v-th operation. Let s_k be the starting floor of p_k.

If $s_k \leq f_j$, then, by Lemma 14, $(k, v) \in view_{s_k - 1} \leq view_{f_j} \leq V_j$. Now suppose that $s_k > f_j$. Thus, p_k must have passed floor f_j before reaching level f_i and invoking IS_{f_i}.

If p_j reads v in $view_{f_j}[k]$, then we are done, as $(k, v) \in view_{f_j}[k] \leq V_j$.

Now suppose that p_j reads an earlier value in v in $view_{f_i}[k]$. By Lemma 14, if p_k reads a non-$\perp$ value in $view_{f_i}$, then it must reads the same value as p_j did. Therefore, to pass floor f_j, p_k must find *false* in all $flag_{f_j}[x]$, $p_x \in W_k$, where W_k is view obtained by p_k from IS_{f_j} (line 10).

In contrast, to return at floor f_j, p_j must have found *true* in some $flag_{f_j}[y]$, $p_y \in W_j$. By the algorithm, p_y wrote *true* to $flag_{f_j}[y]$ *before* invoking IS_{f_j}. Thus, if $p_y \notin W_k$: otherwise, p_k should have also found *true* in $flag_{f_j}[y]$. But W_k and W_j, as outcomes of IS_{f_j}, must be related by containment and, since $p_y \in W_j$ and $p_y \notin W_k$, we have $W_k < W_j$. By the Self-inclusion property of IS, we obtain $(k, v) \in W_j \subseteq V_j$. $\Box_{Lemma\ 17}$

Lemma 18 *Algorithm in Figure 9.6 satisfies the Containment property.*

Proof Let p_i return V_i at floor f_i and p_j return V_j at floor f_j, such that $f_i < f_j$. If $f_i < f_j$, then, by Lemma 17, $V_i \leq V_j$.

Suppose now that $f_i = f_j$. By Lemma 14, p_i and p_j find the same view in $view_{f_i}$. The Containment property of one-shot IS ensures that the views W_i and W_j obtained by p_i and p_j from IS_{f_i} are related by containment, as are V_i and V_j. $\Box_{Lemma\ 18}$

Lemma 19 *Algorithm in Figure 9.6 satisfies the Immediacy property.*

Proof Let p_i return V_i at floor f_i and p_j return V_j at floor f_j. Let $(i, v) \in V_i$ and $(i, v) \in V_j$. We show that $V_i \leq V_j$.

Suppose that $f_i > f_j$. By the condition in line 10, p_i read a value $< v$ in $view_{f_i}[i]$ and, by Lemma 19, p_j read a value $< v$ in $view_{f_j}[i]$—a contradiction.

If $f_i < f_j$, then, by Lemma 17, $V_i \leq V_j$.

If $f_i = f_j$, then, by Lemma 14, they find the same view in $view_{f_i}$ and, by the condition in line 10, $(i, v) \notin view_{f_i}$. Since $(i, v) \in V_j$, we must then have $(i, v) \in W_j$. By the Immediacy property of one-shot IS, we get $W_i \leq W_j$ and, thus, $V_i \leq V_j$. $\Box_{Lemma\ 19}$

Lemma 20 *Algorithm in Figure 9.6 is bounded wait-free with $O(n^3)$ step complexity.*

Proof Suppose that a process p_i starts its v-th IS operation, updates its position in C with v, takes a snapshot of C, and registers the resulting view V at floor $s = |V| = \sum_j V[j]$. If $s \leq n$, we are done, as p_i can only pass through most n floors before, in the worst case, it reaches floor 0 containing view $[0, \ldots, 0]$ and flags $[true, \ldots, true]$. Since each floor involves accessing a one-shot IS instance with $O(n^2)$ step complexity, we get $O(n^3)$ total step complexity.

Suppose now that $s > n$, i.e., there is a non-empty set of processes that invoked at least two operations before time t when p_i took its v-th snapshot of C. Let p_j be the process in this set that was the last to perform its *penultimate* update of C before t, let it happen at time $t' < t$.

As at most n updates of C take place between t' and t, V', the result of the subsequent snapshot of C taken by p_j is such that $|V'| \geq |V| - n$. Since p_j performed exactly one update of C between t' and t, it must have registered V' at its starting floor $s' = |V'| \geq |V| - n = s - n$ and set $flag[j]$ to $true$ before t. The value of p_i in V' is smaller than v and, thus, p_i must return from its v-the operation at level s' or higher, after passing through at most n floors. $\Box_{Lemma\ 20}$

Lemmas 16, 18, 19 and 20 imply:

Theorem 25 *Algorithm in Figure 9.6 is a bounded wait-free implementation of the IS memory with $O(n^3)$ read-write step complexity.*

9.4. Iterated immediate snapshot

We now consider *iterated* shared-memory models. In such models, processes communicate via a series of shared memories M_1, M_2, ... A process proceeds in consecutive rounds 1, 2, ..., and in each round i it accesses memory M_i. In this section, we assume that every memory M_i is an instance of immediate snapshot, and a process simply applies the *update_snapshot*() operation to access it.

Iterated immediate snapshot memory (IIS) is of particular interest for us for two reasons. First, IIS is equivalent to the conventional (non-iterated) read-write shared-memory model, as long as we are concerned with solving distributed tasks or designing non-blocking algorithms (Section 9.4.1). Second, it has a very simple geometric representation, enabling a straightforward characterization of computability (Section 9.4.2).

9.4.1. An equivalence between IIS and read-write

It is straightforward to implement IIS in the read-write shared memory model using the construction in Section 9.1 for each M_i independently.

For the other direction, it is hopeless to look for *wait-free* implementations of the read-write memory in the IIS model in which *every* correct process is able to complete each of its operations. Consider a run in which a correct process p_i is "left behind" in every IIS iteration and, as a result, it never appears in the view of any other process. No write operation performed by p_i in any read-write implementation, based on IIS, will be able to affect read operations performed other processes. Thus, no correct read-write implementation can guarantee that p_i completes any of its writes in that run.

However, as we will show now, IIS can *simulate* read-write memory in a *non-blocking* way. Recall that a non-blocking implementation guarantees that in an infinite execution at least one process *makes progress*. We focus on algorithms in which a process may complete its computation and *terminate* or perform infinitely many reads and writes. Thus, our simulation will guarantee that every correct process either terminates or performs infinitely many (simulated) reads and writes.

We use IIS to implement the read-write model in which memory is organized as a vector of single-writer multiple-reader registers, and every process alternates updates of its dedicated register with atomic snapshots of the memory. Again, we assume that every process runs the full-information protocol: first it writes its input value and every subsequent update includes the outcome of the preceding snapshot.

The implementation maintains, at every process p_i, a local array $c_i[1,\dots,n]$, called a *vector clock*. Each $c_i[j]$ has two components:

- $c_i[j].clock$ that contains the number of update operations of p_j "witnessed" by p_i so far, and
- $c_i[j].val$ that contains the most recent value of p_j's vector clock "witnessed" by p_i so far.

The simulation, presented in Figure 9.7, works as follows. To perform an update, p_i increments $c_i[i].clock$ and sets $c_i[i].clock$ to be the "most recent" vector clock observed so far. To take a memory snapshot, p_i goes through multiple iterations of IIS until the "size" of the currently observed vector clock, $|c_i| = \sum_j c_i[j].clock$, gets "large enough". We explain what we mean by "most recent" and "large enough" below.

In every round of our implementation, p_i writes its current view of the memory and stores an update of it in a local variable $view = view[1],\dots,view[n]$ (line 3). Then for every process p_j, p_i computes the position

$$k = argmax_\ell view[\ell][j].clock$$

and fetches $view[k][j].val$. The resulting vector of the "most recent" values written by the processes is denoted by $top(view)$.

Then p_i checks if $|c| = \sum_j c[j].clock$, the sum of clock values of all the processes equals the current round number. Intuitively, the condition that the currently simulated snapshot of p_i contains all the most recent written values and relates by containment to the results of all other simulated snapshot operations. Indeed, as the clock values grow monotonically, snapshots S and S' produced in IIS rounds r and r', $r \leq r'$, satisfy $S \leq S'$.

Formally, every process p_i goes through a number of *phases*, where phase $k = 1, 2, \dots$ starts when p_i's local variable $c_i[i].clock$ is assigned value k (line 1 for $k = 1$ or line 11 for $k > 1$). Phase k ends when p_i departs after executing line 8 or starts phase $k+1$. The argument of the write operation of phase k is the value of $c[i].val$ initialized at the end of phase $k-1$ in line 10 if $k > 1$ and the input value of p_i otherwise. The outcome of the k-th simulated snapshot operation is chosen to be the last value of $c.val$ computed in line 5 of the phase.

To justify that our simulation is correct, we first prove a few auxiliary lemmas. Let $view_i^r$ and c_i^r denote, respectively, the view and the clock vector evaluated by process p_i in round r, i.e., in lines 4 and 5, respectively, of the rth iteration of the algorithm. We say that $c_i^r \leq c_j^r$ if $\forall k: \; c_i^r[k].clock \leq c_j^r[k].clock$, i.e., c_i^r contains at least as recent perspective on the simulated state as c_j^r. Recall that $|c_i^r| = \sum_j c_i^r[k].clock$.

Lemma 21 *For all $r \in \mathbb{N}$, $p_i, p_j \in \Pi$, $|c_i^r| \leq |c_j^r|$ implies $c_i^r \leq c_j^r$.*

```
Shared variables: IS memories $IS_1, IS_2, \dots$

Local variables at each $p_i$: $c_i[1,\dots,n]$, initially $[\bot,\dots,\bot]$

Code for process $p_i$:
(1)  $r := 0$; $c[i].clock := 1$; $c_i[i].val :=$ input of $p_i$;      { memorize $p_i$'s input }
(2)  repeat forever
(3)       $r := r+1$
(4)       $view := IS_r.update\_snapshot(c)$      { update the view using $IS_r$ }
(5)       $c := top(view)$      { update the clock vector with the most recent information }
(6)       if $|c| = r$ then      { if the current snapshot is complete }
(7)            if $decided(c.val)$ then      { if ready to decide }
(8)                 return $decision(c.val)$
(9)            endif
(10)           $c_i[i].val := c$      { compute the next value to write }
(11)           $c_i[i].clock := c_i[i].clock + 1$      { update the local clock }
(12)      endif
(13) end repeat
```

Figure 9.7.: Implementing AS using IIS

Proof By the Set Inclusion property of IS (see Section 9.1), the views evaluated by p_i and p_j in line 4 of round r are related by containment, i.e., $view_i^r \subseteq view_j^r$ or $view_j^r \subseteq view_i^r$. Since c_i^r and c_j^r are computed as the vector of the most up-to-date values gathered from the views (line 5), we have $c_i^r \leq c_j^r$ or $c_j^r \leq c_i^r$.

Suppose, by contradiction that $|c_i^r| \leq |c_j^r|$ but $c_i^r \not\leq c_j^r$, i.e., $c_j^r \leq c_i^r$ but $c_j^r \neq c_i^r$. Since the operation $|c|$ sums up the values of $c[i].clock$, we get $|c_j^r| > |c_i^r|$—a contradiction. Thus, $|c_i^r| \leq |c_j^r|$ indeed implies $c_i^r \leq c_j^r$. $\square_{Lemma\ 21}$

Since, by Lemma 21, $|c_i^r| = |c_j^r|$ implies $c_i^r = c_j^r$, we have:

Corollary 2 *All processes that complete a snapshot operation in round r, evaluate the same clock vector c, $|c| = r$.*

Lemma 22 *For all $r \in \mathbb{N}$, $p_i \in \Pi$, $|c_i^r| \geq r$.*

Proof By the Self-inclusion property of IS, $c_1^1[i].clock = 1$, and, thus, $|c_1^1| \geq 1$. Suppose, inductively, that for all p_i, $|c_i^r| \geq r$ for some $r \geq 1$.

Since the view computed by p_i in round r is written afterward to IS_{r+1}, the values of $|c_i^r|$ do not decrease with r. Thus, if $|c_i^r| > r$, then $|c_i^{r+1}| \geq |c_i^r| \geq r+1$. On the other hand, if $|c_i^r| = r$, i.e., p_i completes its snapshot operation in round r, then p_i increments $c_i[i].clock$ and we have $|c_i^{r+1}| > |c_i^r| + 1 \geq r+1$. In both cases, $|c_{r+1}^r| \geq r+1$ and the claim follows by induction. $\square_{Lemma\ 22}$

The values of $c_i^r.clock$ can only increase with r. Thus, by Lemmas 21 and 22, we have:

Corollary 3 *If $|c_i^r| = r$ (i.e., p_i completes a snapshot operation in round r), then for all p_j and $r' > r$, we have $c_i^r \leq c_j^{r'}$.*

Now we show that some correct process always makes progress in the simulated run. We say that a process *terminates* once it reaches line 8. Note that if a process terminates in round r, it does not access any $IS_{r'}$, for $r' > r$.

Lemma 23 *For all $r \in \mathbb{N}$, if there is a correct process reaches round r, eventually some correct non-terminating process its current phase in round $r' \geq r$.*

Proof By contradiction, assume that there is an execution in which some correct non-terminated process is in round r and no correct non-terminated process ever completes its current phase, i.e., no process p_i ever increases the value of $c_i[i].clock$. Thus, there exists a clock vector c such that $\forall r' \geq r$, $p_i \in \Pi$: $c_i^{r'} = c$.

By Lemma 22, for all p_i and $r' \geq r$, $|c| = |c_i^r| \geq r$. Consider round $r' = |c| \geq r$. By the assumption, every correct non-terminated process p_i evaluates $c_i^{r'} = c$ and, by the algorithm, terminates in round r'—a contradiction. $\Box_{Lemma\ 23}$

Now we are ready to prove correctness of our simulation.

Theorem 26 *Every run R simulated by the algorithm in Figure 9.7 is indistinguishable from a run R_s of the full information protocol in the AS model in which either every correct (in R) process terminates or some correct process takes infinitely many steps.*

Proof Given R, we construct R_s as follows. Assuming that p_i completes its kth phase in r, let W_i^k and S_i^k denote, respectively, the corresponding simulated update and snapshot operations. First we order all resulting S_i^k according to the round numbers in which they were completed. Then we place each W_i^k just before the first snapshot that contains the kth simulated view of p_i.

By Corollary 2, all snapshot outcomes produced in the same round are identical. By Corollary 3, snapshot outcomes grow with the round numbers. Thus, in R_s, every two snapshots are related by containment, and every next snapshot is a copy or a superset of the previous one. Furthermore, the Self-inclusion property of one-shot IS instances used in the algorithm implies that every S_i^k contains the kth simulated view of p_i. Thus, in R_s, every p_i executes the operations appear in the order they take place in R: W_i^1, S_i^1, W_i^2, S_i^2, $\ldots$.

By construction, the outcome of every S_i^r contains the most recent written value for each process. $\Box_{Theorem\ 26}$

Now suppose that a given distributed task is solvable in the AS model: in every run, every process eventually reaches a *decided* state, captured in line 7 of our algorithm.

Assuming, without loss of generality, that a decided process simply stops taking steps, our non-blocking solution brings the next correct process to the output, then the next one, etc., until every correct process outputs. Note that there is no loss of generality in assuming that a process stops after producing an output, since it juts corresponds to the execution in which the process crashes just after deciding.

Therefore, Theorem 26 implies that IIS is equivalent to AS (or, more generally the read-write model) in terms of task solving:

Corollary 4 *A task is solvable in IIS if and only if it is solvable in the read-write asynchronous model.*

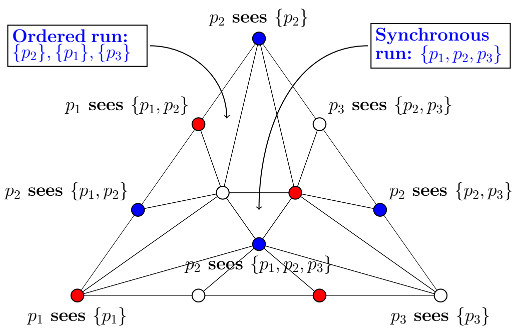
Note that in the above prove is that we do not use the Immediacy property of IS. Thus, the simulation would still be correct even if we replace $view := IS_r.update\_snapshot(c)$ in line 4 with $AS_r.update(c)$ followed by $view := AS_r.snapshot(c)$.

9.4.2. Geometric representation of IIS

The IIS model allows for a simple geometric representation. All possible runs of one round of IIS can be represented as a *standard chromatic subdivision* of the $(n-1)$*-dimensional simplex.*

The example depicted in Figure 9.8 describes the views obtained by three processes, p_1, p_2, and p_3, after each executes For example, the blue corner of the triangle models the view of p_1 in a run where it only sees itself. The internal points on the blue-green face model the views of p_1 and p_2 in runs where they see each other but miss p_3. Finally, the internal points of the triangle model the views of the processes in which they see all three. A triangle in the subdivision models the set of views that can be obtained in the same run.

As we can see, the resulting views and runs result in a nice *simplicial complex* that is simply a subdivision of the triangle corresponding to the initial state of the system. Multiple rounds of the IIS model can thus be represented as an *iterated* standard chromatic subdivision, where each of the triangles is subdivided, then each of the resulting triangles is subdivided, etc.



Ordered run: $\{p_2\}, \{p_1\}, \{p_3\}$
p_2 sees $\{p_2\}$
Synchronous run: $\{p_1, p_2, p_3\}$
p_1 sees $\{p_1, p_2\}$
p_3 sees $\{p_2, p_3\}$
p_2 sees $\{p_1, p_2\}$
p_2 sees $\{p_2, p_3\}$
p_2 sees $\{p_1, p_2, p_3\}$
p_1 sees $\{p_1\}$
p_3 sees $\{p_3\}$

Figure 9.8.: One round of 3-process IIS as a standard chromatic subdivision of a chromatic 2-simplex: red vertices model possible resulting states of p_1, blue–p_2, and white–p_3.

Notice that one round of the (full-information) AS model produces runs that do not fit the subdivision depicted in Figure 9.8. For example, the AS model allows a run in which p_1 only sees itself and p_2, but both p_2 and p_3 see all three processes. In Figure 9.8 this runs corresponds to the triangle formed by the blue vertex on the face (p_1, p_2) and the green and read vertices in the interior that overlaps with other triangles in the subdivision. But since this run does not satisfy the Immediacy property of IS, it is excluded by the IS model.

The fact that one round of the IS model is captured by the subdivision depicted in Figure 9.8 is obvious for three processes. More generally, to model runs of the IIS model in a system of n processes, consider the initial system state s represented as $(n-1)$-dimensional *chromatic simplex* s, i.e., a set of n vertices, each vertex corresponding to a distinct process. $Chr\mathsf{s}$ is now defined inductively on the dimension of s.

If s is zero dimensional, which corresponds to a system of only one process, we let $Chr\mathsf{s} = \mathsf{s}$. Suppose now, inductively, that s has dimension $n-1$, and that we already took the chromatic subdivision of its $(n-2)$-*skeleton*, i.e., all subsets of size at most $n-1$. Take a new $(n-1)$-simplex s'. For each face $\mathbf{t}$ of s, let $\bar{\mathbf{t}}'$ be the *complementary face* of s', that is, the face of s' corresponding to the processes that do not appear in $\mathbf{t}$. Then every simplex consisting of the vertices $\bar{\mathbf{t}}'$ and the vertices of any simplex in the chromatic subdivision of $\mathbf{t}$ is added to the resulting *simplicial complex* $Chr\mathsf{s}$. If we iterate this construction k times we obtain the kth chromatic subdivision, $Chr^k C$.

Bibliographic notes

Borowsky and Gafni [12] introduced the notion of immediate snapshot (IS) and gave the first one-shot IS implementation for the read-write model.

The task of renaming was originally posed and solved by Attiya et al. [6] for the message-passing model. The adaptive renaming algorithm in Figure 9.3 is due to Attiya and Welch [10, Chapter 16] who adapted the algorithm by Attiya et al. [6] to the read-write shared-memory model. Attiya, Fouren and Gafni [8] claimed that this algorithm and several alternative algorithms published at the time expose exponential (in p) read-write step complexity in some executions. The $O(p^3)$ renaming algorithm described in Figure 9.5 was proposed by Borowsky and Gafni [12].

The long-lived IS simulation described in Section 9.3 is a simplified and slightly corrected version of the adaptive simulation by Attiya, Fouren and Gafni [7]. The IIS-based simulation of the conventional read-write model presented in Section 9.4 is due to Gafni and Rajsbaum [29].

The proof that Chrs is indeed a subdivided simplex was sketched by Linial [52] and independently found Kozlov [44]. A thorough discussion on the combinatorial methods in distributed computing can be found in [36].

Exercises

1. Show that the IS object does not have a sequential specification.

2. Suppose that k processes accessed a one-shot IS objects and obtained sets of distinct sizes $1 \leq \ell_1 < \ldots < \ell_s$.

   Show that $\ell_s = k$ and if $s \leq 2$, then for all $j = 2 \ldots, s$, the number of processes that obtained a set of size ℓ_j is $\ell_j - \ell_{j-1}$.

3. Assuming the full information protocol, show that the IS model is stronger that the AS model: every run of the IS model can be represented as a run of AS model.

4. Prove that the algorithm described in Figure 9.3 is correct. Will it remain correct if we replace $update$ and $snapshot$ with $store$ and $collect$, respectively.

5. Does the AS-based renaming algorithm in Figure 9.3 have a run in which n processes output names $1, 2, \ldots, n$? What about the IS-based algorithm in Figure 9.5?

Part IV.

Consensus objects

10. Consensus and universal construction

In the first part of this book, we considered multiple powerful abstractions that can be implemented, in the wait-free manner, from read-write registers. In this chapter, we address a more general question:

> Given object types T and T', is there a wait-free implementation of an object of type T from objects of type T'?

We define a fundamental *consensus* object type and show that consensus objects are *universal*: any object type can be implemented, in the wait-free manner, using read-write registers *and* consensus objects. In the next chapter, we show that read-write register cannot, by themselves, implement a wait-free consensus object shared by 2 processes and, thus, are not universal even in a system of 2 processes. This observation brings the notion of a *consensus number* of a given object type: the maximal number of processes in which the type is universal.

Overall, in this chapter we give a definition of consensus and demonstrate its power in implementing arbitrary object types. In the next chapter, we discuss the downside of this abstraction, namely, the difficulty of its implementations.

10.1. Consensus object: specification

The *consensus* object type exports an operation $propose()$ that takes one input parameter v in a *value set* V ($|V| \geq 2$) and returns a value in V. Let $\perp$ denote a default value that cannot be proposed by a process ($\perp \notin V$). Then $V \cup \{v\}$is the set of states a consensus object can take, $\perp$ is its initial state, and its sequential specification is defined in Figure 10.1. A consensus objects can thus be seen as a "write-once" register that keeps forever the value proposed by the first $propose()$ operation. Then, any subsequent $propose()$ operation returns the first written value.

Given a *linearizable* implementation of the consensus object type, we say that a process *proposes* v if it invokes $propose(v)$ (we then say that it is a *participant* in consensus). If the invocation of $propose(v)$ returns a value v', we say that the invoking process *decides* v', or v' is decided by the consensus object. We observe now that any execution of a *wait-free* linearizable implementation of the consensus object type satisfies three properties:

- *Agreement:* no two processes decide different values.
- *Validity:* every decided value was previously proposed.

  Indeed, otherwise, there would be no way to linearize the execution with respect to the sequential specification in Figure 10.1 which only allows to decide on the first proposed value.

```
operation propose($v$):
    if ($x = \perp$) then $x := v$ endif;
    return ($x$).
```

Figure 10.1.: Sequential specification of consensus

- *Termination:* Every correct process eventually decides.

  This property is implied by wait-freedom: every process taking sufficiently many steps of the consensus implementation must decide.

10.2. A wait-free universal construction

In this section, we show that if, in a system of n processes, we can wait-free implement consensus, then we can implement *any* total object type.

Recall that a total object type can be represented as a tuple (Q, q_0, O, R, δ), where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set of responses, and δ is a binary relation on $O \times Q \times R \times Q$, total on $O \times Q$: $(o, q, r, q') \in \delta$ if operation o is applied when the object's state is q, then the object *can* return r and change its state to q'. Note that for *non-deterministic* object types, there can be multiple such pairs (r, q') for given o and q.

The goal of our universal construction is, given an object type $\tau = (Q, O, R, \delta)$, to provide a wait-free linearizable implementation of τ using read-write registers and atomic consensus objects.

10.2.1. Deterministic objects

For deterministic object types, δ can be seen as a function $O \times Q \rightarrow R \times Q$ that associates each state an operation with a unique response and a unique resulting state. The state of a deterministic object is thus determined by a sequence of operations applied to the initial state of the object. The universal construction of an object of a deterministic type is presented in Figure 10.2.

Every process p_i maintains a local variable $\mathit{linearized}_i$ that stores a sequence of operations that are executed on the implemented object do far. Whenever p_i has a new operation op to be executed on the implemented object it "registers" op in the shared memory using a collect object R. As long as p_i finds new operations that were invoked (by p_i itself or any other process) but not yet executed in R, it tries to agree on the order in which operations must be executed using the "next" consensus object $C[k_i]$ that was not yet accessed by p_i. If the set of operations returned $C[k_i]$ contains op, p_i deterministically computes the response of op using the specification of the implemented object and $\mathit{linearized}_i$. Otherwise, p_i proceeds to the next consensus object $C[k_i + 1]$.

Intuitively, this way the processes make sure that their perspectives on the evolution of the implemented object's state are mutually consistent.

Correctness.

Lemma 24 *At all times, for all processes p_i and p_j, $\mathit{linearized}_i$ and $\mathit{linearized}_j$ are related by containment.*

Proof We observe that each $\mathit{linearized}_i$ is constructed by adding the batches of requests decided by consensus objects $C_1, C_2, \ldots$, in that order. The agreement property of consensus (applied to each of these consensus objects) implies that, for each p_j, either $\mathit{linearized}_i$ is a prefix of $\mathit{linearized}_j$, or vice versa. $\square_{\mathit{Lemma}\ 24}$

Lemma 25 *Every operation returns in a finite number of its steps.*

Proof Suppose, by contradiction, that a process p_i invokes an operation op and executes infinitely many steps without returning. By the algorithm, p_i forever blocks in the repeat-until clause in lines 8-14. Thus, p_i proposes batches of requests containing its request (op, i, seq_i) to an infinite sequence of

```
Shared objects:
    $R$, collect object, initially $\perp$
    $C_1, C_2, \ldots$, consensus objects

Local variables, for each process $p_i$:
    integer $seq_i$, initially 0          { the number of executed requests of $p_i$ }
    integer $k_i$, initially 0          { the number of batches of executed requests }
    sequence $linearized_i$, initially empty          { the sequence of executed requests }

Code for operation $op$ executed by $p_i$:
6   $seq_i := seq_i + 1$
7   $R.store(op, i, seq_i)$          { publish the request }
8   repeat
9       $V := R.collect()$          { collect all current requests }
10      $requests := V - \{linearized_i\}$          { choose not yet linearized requests }
11      $k_i := k_i + 1$
12      $decided := C[k_i].propose(requests)$
13      $linearized_i := linearized_i.decided$          { append decided requests }
14  until $(op, i, seq_i) \in linearized_i$
15  return the result of $(op, i, seq_i)$ in $linearized_i$ using $\delta$ and $q_0$
```

Figure 10.2.: Universal construction for deterministic objects

consensus instances $C_1, \ldots$ but the decided batches never contain (op, i, seq_i). By validity of consensus, there exists a process $p_j \neq p_i$ that accesses infinitely many consensus objects. By the algorithm, before proposing a batch to a consensus object, p_j first collects the batches currently stored by other processes in a collect object R. Since p_i stores its request in R and never updates it since that, eventually, every such process p_j must collect the p_i's request and propose it to the next consensus object. Thus, every value returned by the consensus objects from some point on must contain the p_i's request—a contradiction. $\square_{Lemma\ 25}$

Theorem 27 *For each type $\tau = (Q, q_0, O, R, \delta)$, the algorithm in Figure 10.2 describes a wait-free linearizable implementation of τ using consensus objects and atomic registers.*

Proof Let H be the history an execution of the algotihm in Figure 10.2. By Lemma 24, local variables $linearized_i$ are prefixes of some sequence of requests $linearized$. Let L be the legal sequential history, where operations and are ordered by $linearized$ and responses are computed using q_0 and δ. We construct H', a completion of H, by adding responses to the incomplete operations in H that are present in L. By construction, L agrees with the local history of H' for each process.

Now we show that L respects the real-time order of H. Consider any two operations op and op' such that $op \rightarrow_H op'$ and suppose, by contradiction that $op' \rightarrow_L op$. Let (op, i, s_i) and (op', j, s_j) be the corresponding requests issued by the processes invoking op and op', respectively. Thus, in $linearized$, (op', j, s_j) appears before (op, i, s_i), i.e., before op terminates it witnesses (op', j, s_j) being decided by consensus objects $C_1, C_2, \ldots$ before (op', j, s_j). But, by our assumption, $op \rightarrow_H op'$ and, thus, (op', j, s_j) has been stored in the collect object R *after* op has returned. But the validity property of consensus does not allow to decide a value that has not yet been proposed—a contradiction. Thus, $op \rightarrow_L op'$, and we conclude that H is linearizable. $\square_{Theorem\ 27}$

```
Shared objects:
    $R$, collect object, initially $\perp$        { *published requests* }
    $C_1, C_2, \ldots$, consensus objects
    $S$, collect object, initially $(1, \epsilon)$        { *the current consensus object and the last committed sequence of requests* }

Local variables, for each process $p_i$:
    integer $seq_i$, initially 0        { *the number of executed requests of $p_i$* }
    integer $k_i$, initially 0        { *the number of batches of executed requests* }
    sequence $linearized_i$, initially $\epsilon$        { *the sequence of executed requests* }

Code for operation *op* executed by $p_i$:
16  $seq_i := seq_i + 1$
17  $R.store(op, i, seq_i)$        { *publish the request* }
18  $(k_i, linearized_i) := \max(S.collect())$        { *get the current consensus object and the most recent state* }
19  **repeat**
20      $V := R.collect()$        { *collect all current requests* }
21      $requests := V - \{linearized_i\}$        { *choose not yet linearized requests* }
22      $decided := C[k_i].propose(requests)$
23      $linearized_i := linearized_i.decided$        { *append decided requests* }
24      $k_i := k_i + 1$
25  **until** $(op, i, seq_i) \in linearized_i$
26  $S.store((k_i + 1, linearized_i))$        { *publish the current consensus object and state* }
27  return the result of $(op, i, seq_i)$ in $linearized_i$ using $\delta$ and $q_0$
```

Figure 10.3.: Bounded wait-free universal construction for deterministic objects

10.2.2. Bounded wait-free universal construction

The implementation described in Figure 10.2 is wait-free but not *bounded* wait-free. A process may take arbitrarily many steps in the repeat-until clause in lines 8-14 to "catch up" with the current consensus object.

It is straightforward to turn this implementation into a bounded wait-free. Before returning an operation's response (line 15), a process posts in the shared memory the sequence of requests it has witnessed committed together with the id of the last consensus object it has accessed. On invoking an operation, a process reads the memory to get the "most recent" state on the implemented object and the "current" consensus id. Note that multiple processes concurrently invoking different operations might get the same estimate of the "current state" of the implementation. In this case only one of them may "win" in the current consensus instance and execute its request. But we argue that the requests of "lost" processes must be then committed by the next consensus object, which implies that every operation returns in a bounded number of its own steps.

The resulting implementation is presented in Figure 10.3.

To prove the following theorem, we recall that collect objects R and S can be implemented with $O(n)$ read-write step complexity (Chapter 8).

Theorem 28 *For each type $\tau = (Q, q_0, O, R, \delta)$, the algorithm in Figure 10.3 describes a wait-free linearizable implementation of τ using consensus objects and atomic registers, where every operation returns in $O(n^2)$ shared-memory steps.*

Proof As before, all invoked operations are ordered in the same way using a sequence of consensus objects, so the proof of linearizability is similar the one of Theorem 27.

To prove bounded wait-freedom, consider a request (op, i, ℓ) issued by a process p_i. By the algorithm, p_i first publishes its request and obtains the current state of the implemented object (line 18), denoted k and s, respectively. Then p_i proposes all requests it observes to be proposed but not yet committed to consensus object C_k. If (op, i, ℓ) is committed by C_k, then p_i returns after taking $O(n)$ read-write steps (we assume that both collect operations involve $O(n)$ read-write steps).

Suppose now that (op, i, ℓ) is not committed by C_k. Thus, another process p_j has previously proposed to C_k a set of requests that did not include (op, i, ℓ). Thus, p_j collected requests in line 20 before or concurrently with the store operation in which p_i published (op, i, ℓ) (line 17). Moreover, p_j did not store the result of its operation in S (line 26) before p_i performed its collect of S in line 18. The situation may repeat when p_i proceeds to consensus object C_{k+1}, but only if there is another process p_k that previously "won" C_{k+1} with a sequence not containing (op, i, ℓ), but has not yet stored its state in S. Note that p_k must be different from p_j, otherwise , p_j would store $k_i + 1$ in S before collecting R which, as (op, i, ℓ) was not found in R by p_j should have happened before of concurrently with the store in S performed by p_i.

There can be at most $n-1$ processes that may prevent p_i from "winning" consensus objects and, thus, p_i may perform at most $n-1$ iterations in lines 19-25. As each iteration consists of $O(n)$ shared-memory steps, we get $O(n^2)$ step complexity for individual operations. $\square_{Theorem\ 28}$

10.2.3. Non-deterministic objects

The universal construction in Figure 10.2 assumes the object type is deterministic, where for each state and each operation there exists exactly one resulting state and response pair. Thus, given a sequence of request, there is exactly one corresponding sequence of responses and state transitions.

A "dumb" way to use our universal construction is to consider any deterministic restriction of the given object type. But this may not be desirable if we expect the shared object to behave probabilistically (e.g., in randomized algorithms). A "fair" non-deterministic universal construction can be derived from the algorithm in Figure 10.3 as follows. Instead of only proposing a sequence of requests in line 22, process p_i (using a local random number generator) proposes a sequence of responses and state transitions corresponding to a sequence of operations $requests$ applied to the last state in $linearized_i$. One of the proposed sequences of responses and state transitions will "win" the consensus instance and will be used to compute the new object state.

10.3. Bibliographic notes

The "Byzantine generals" problem, consisting in reaching agreement in a synchronous system of processes subject to Byzantine (arbitrary) failures, was introduced by Lamport, Shostak and Pease [61, 50]. Fisher, Lynch, and Paterson considered the problem of reaching agreement in asynchronous crash-prone systems and introduced the notion of consensus.

Universality of consensus is inspired by the replicated state machine approach proposed by Lamport [48] and elaborated by Schneider [64]. The consensus-based universal construction that gives a wait-free implementation of any (total) sequential type was proposed by Herlihy [35]. Hadzilacos and Toueg defined a closely related abstraction of *total-order broadcast* and showed that it is equivalent to consensus (assuming reliable communication media) [32].

Exercises

1. Show that the two definitions of consensus given in Section 10.1 are *equivalent*: a wait-free linearizable consensus object (Figure 10.1) satisfies the properties of Agreement, Validity and Termination and, vice versa, any algorithm using atomic base objects satisfying these three properties is a wait-free linearizable consensus implementation.

2. Find an algorithm solving the relaxation of consensus in which only two out of the three properties are satisfied.

3. Show that the algorithm described in Figure 10.2 is not *bounded* wait-free.

11. Consensus number and the consensus hierarchy

In the previous chapter, we introduced a notion of a *universal* object type. Using read-write registers and objects of a universal type and, one can implement an object of any total type in the wait-free manner. As we have shown, one example of a universal type is *consensus*. Therefore, the power of an object type can be measured by the ability of its objects to implement consensus.

We show in this section that atomic registers cannot implement a consensus object shared by two processes, thus, the register type is not universal even in a system of two processes. If, however, in addition to registers, we may use queue objects, then we can implement 2-processe consensus, but not 3-process consensus.

More generally, we introduce the notion of *consensus number* of an object type T, the largest number of processes for which T is universal. Consensus numbers are fundamental in capturing the relative power of object types, and we show how to evaluate the consensus power of various object types.

11.1. Consensus number

The *consensus number* of an object type T, denoted by $cons(T)$, is the *highest* number n such that it is possible to wait-free implement a consensus object from atomic registers and objects of type T, in a system of n processes. If there is no such largest n, i.e., consensus can be implemented in a system of arbitrary number of processes, the consensus number of T is said to be infinite.

Note that if there exists a wait-free implementation of an object in a system of n process implies a wait-free implementation in a system of any $n' < n$ processes. Thus, the notion of consensus number is well-defined. By the definition, if $cons(T) < cons(T')$, then there is no wait-free implementation of an object of type T' from objects of type T and registers in a system of $cons(T) + 1$ or more processes.

If atomic registers are strong enough to wait-free implement consensus for any number of processes, i.e., $cons(regiter) = \infty$, then all object types would have the same consensus number, and the very notion of consensus number would be useless. We show below that this is not the case. Moreover, we show that for each n, there exists object types T, such that $cons(T) = n$, i.e., the *consensus hierarchy* is populated for each level n.

11.2. Preliminary definitions

In this section, we introduce some machinery that is going to be used to compute consensus numbers of object types. Let us consider an algorithm A that implements a wait-free consensus object assuming that processes only propose values 0 and 1, we call it a *binary consensus* object.

11.2.1. Schedule, configuration and valence

We consider a system in which n sequential processes communicate by invoking operations on "base" atomic (linearizable) objects of types $T_1, \ldots, T_x$. As the base objects are atomic, an execution in this system can be modeled by a sequential history that (1) includes all the operations on base objects issued by the processes (except possibly the last operation of a process if that process crashes), (2) is legal with respect to the type of each base object, and (3) respects the real time occurrence order on the operations. Recall that this sequential history is called a *linearization*.

Schedules and configurations A *schedule* is a sequence of base-object operations. In the following, we assume that the base object types are deterministic and the processes are running deterministic wait-free consensus algorithms. Thus, we can represent an operation in a schedule only by the identifier of the process that issues that operation.

A *configuration* C is a global state of the system execution at a given point in time. It includes the state of each base object plus the local state of each process. The configuration $p(C)$ denotes the configuration obtained from C by applying an operation issued by the process p. More generally, given a schedule S and a configuration C, $S(C)$ denotes the configuration obtained by applying to C the sequence of operations defining S.

In an *input* configuration of algorithm A, base objects and processes are in their initial states. In particular, for binary consensus, the initial state of a process can be 0 or 1, depending on the value the process is about to propose.

Valence The notion of *valence* is fundamental in proving consensus impossibility results. Let C be a configuration resulting after a finite execution of algorithm A.

We say that configuration C is v-*valent* if every schedule applied to C leads to v as the decided value. We say that v is the valence of that configuration C. A 0-valent or 1-valent configuration is said to be *monovalent*. A configuration that is not monovalent is said to be *bivalent*.

By the definition, every descendant $S(C)$ of a monovalent configuration C must be monovalent. Similarly, if a configuration C has a bivalent descendant $S(C)$, then C is bivalent.

Lemma 26 *Every configuration of a wait-free consensus implementation A is monovalent or bivalent.*

Proof Let $S(C)$ a configuration of A reachable from an initial configuration C by a finite schedule S. Since the algorithm is wait-free, for any sufficiently long S', some process must decide in $S'(S(C))$. Since only 0 and 1 can be proposed and, thus, decided, the set of values that can be decided in extensions of $S(C)$ is a non-empty subset of $\{0, 1\}$. $\square_{Lemma\ 26}$

Lemma 27 *A configuration in which a process decides is monovalent.*

Proof By Lemma 26, if a configuration Suppose, by contradiction, that a process p decides $v \in \{0, 1\}$ in a bivalent configuration $S(C)$. Since C is bivalent, there exists a schedule $S'(S(C))$ in which value $1 - v$ is decided, contradicting the agreement property of consensus. $\square_{Lemma\ 27}$

The corollary of Lemmas 26 and 27 is that no process can decide in a bivalent configuration.

11.2.2. Bivalent initial configuration

Our next observation is that any wait-free consensus algorithm must have a bilent *initial* configuration C. In other words, for some distribution of input values, the decided value may depend on the schedule: in some $S(C)$, 0 is decided and in some $S'(C)$, 1 is decided.

Lemma 28 *Any wait-free consensus implementation for 2 or more processes has a bivalent initial configuration.*

Proof Let C_0 be the initial configuration in which all the processes propose 0, and C_i, $1 \leq i \leq n$, the initial configuration in which the processes from p_1 to p_i propose the value 1, while all the other processes propose 0. So, all the processes propose 1 in C_n. Thus, any two adjacent configurations C_{i-1} and C_i, $1 \leq i \leq n$, differ only in p_i's proposed value: p_i proposes 0 in C_{i-1} and 1 in C_i. Moreover, it follows from the validity property of consensus and Lemma 26, that C_0 is 0-valent and C_n is 1-valent.

Let us assume that all configurations $C_0, \dots, C_n$ are monovalent. As $n \geq 2$, there are two consecutive configurations C_{i-1} and C_i, such that C_{i-1} is 0-valent and C_i is 1-valent.

Since the algorithm is wait-free, for any sufficiently long schedule S, some process p_j decides in $S(C_{i-1})$, and, since C_{i-1} is 0-valent, the decided value must be 0. Let us suppose that p_i takes no steps in S.

But as every process besides p_i has the same inputs in C_{i-1} and C_i and the states of base objects in the two initial configurations are identical, no process besides p_i can distinguish $S(C_{i-1})$ and $S(C_i)$. Thus, p_j must also decide 0 in $S(C_i)$, contradicting the assumption that C_i is 1-valent. $\square_{Lemma\ 28}$

Note that the proof above would work even if we assume that *at most one* process may initially crash. In particular, if p_i crashes before taking any step, then no other process can distinguish an execution starting from C_{i-1} from an execution starting from C_i.

11.2.3. Critical configurations

We now show that every wait-free consensus algorithm for two or more processes has a *critical* configuration D with the following properties:

- D is bivalent;
- for every process p_i, $p_i(D)$ is monovalent;
- there exists an object X, such that every process p_i is about to access X in its next step in D.

In other words, one step of any given process applied to a critical configuration determines the decision value.

Lemma 29 *Any wait-free consensus implementation A for 2 or more processes has a critical configuration.*

Proof By Lemma 28, A has a bivalent initial configuration C. We are going to prove that C has a critical descendant $S(C)$.

Suppose not, i.e., for every schedule S, there exists p_i such that $p_i(S(C))$ is bivalent. Therefore, starting from C, we inductively construct an infinite schedule $\tilde{S}$ that, when applied to C, only goes through bivalent configurations: for every its prefix S, $S(C)$ is bivalent. Indeed, let q_1 be any process such that $q_1(C)$ is bivalent, q_2 be any process such that $q_2(q_1(C))$, etc. Then, by Lemma 27, starting from C, the resulting infinite schedule $\tilde{S} = q_1, q_2, \dots$ can never reach a configuration in which a process decides—a contradiction with the assumption that A is a wait-free consensus algorithm.

Thus, C has a bivalent descendant configuration D such that for every p_i, $p_i(D)$ is monovalent.

Now suppose, by contradiction, that there exist two processes p and q that access different objects in their next steps enabled in D. We can safely assume that $p(D)$ is 0-valent and $q(D)$ is 1-valent. We encourage the reader to see why this is the case.

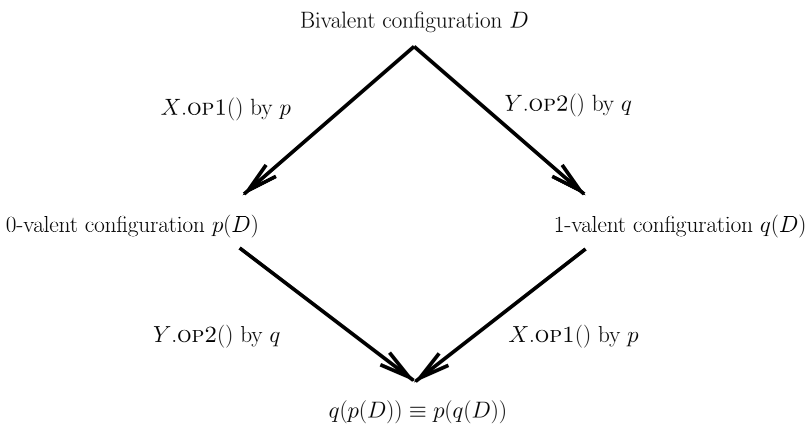


Figure 11.1.: Operations issued on distinct objects

Then the steps of p and q applied to D *commute*, i.e., $q(p(D))$ and $p(q(D))$ are *identical*: in the two configurations, base-objects states and process states are the same (Figure 11.1).

Since $p(D)$ is 0-valent, $q(p(D))$ is 0-valent, and since $q(D)$ is 1-valent, $p(q(D))$ is 1-valent—a contradiction.

Thus, D is indeed a critical configuration of algorithm A. $\square_{Lemma\ 29}$

Note that Lemma 29 holds for *any* wait-free consensus algorithm. By analyzing steps that processes can apply to a critical configuration and using the number of available processes, we can deduce the consensus number of any given object type.

11.3. Consensus number of atomic registers

Atomic registers are fundamental objects in concurrent shared-memory systems. In this section, we show that they are however too weak to solve wait-free consensus even for two processes. Put differently, the consensus number of object type atomic register is 1.

Theorem 29 *There does not exist a wait-free consensus implementation for two processes from atomic registers.*

Proof By contradiction, suppose that there exists a wait-free consensus algorithm A for two processes, p and q, using atomic registers. By Lemma 29, A has a critical configuration D, i.e., D is bivalent, $p(D)$ and $q(D)$ are monovalent, and the two processes are about to access the same register R in their next steps enabled in D. Since $p(D)$ and $q(D)$ are the only two one-step descendants of D, it must hold that $p(D)$ and $q(D)$ have different valences. Without loss of generality, assume that $p(D)$ is 0-valent and $q(D)$ is 1-valent.

Let OP_1 and OP_2 be base-object operations performed by, respectively, processes p and q in their next steps enabled in configuration D.

The following cases are then possible:

- OP_1 and OP_2 are read operations

As a read operation on an atomic register does not modify its value, this case is the same as the previous one where p and q access distinct registers.

- One of the two operations $\textsc{op}_1$ and $\textsc{op}_2$ is a write. Without loss of generality, suppose that q is about to write in R in D (Figure 11.2).

  Consider configurations $q(p(D)$ and $q(D)$. Since p accessed R in $\textsc{op}_1$ and q writes in R in $\textsc{op}_2$, the state of D is the same in the two configurations. Thus, the only difference between the two is the local state of p: p took one more step after D in $q(p(D)$, but not in $q(D)$.

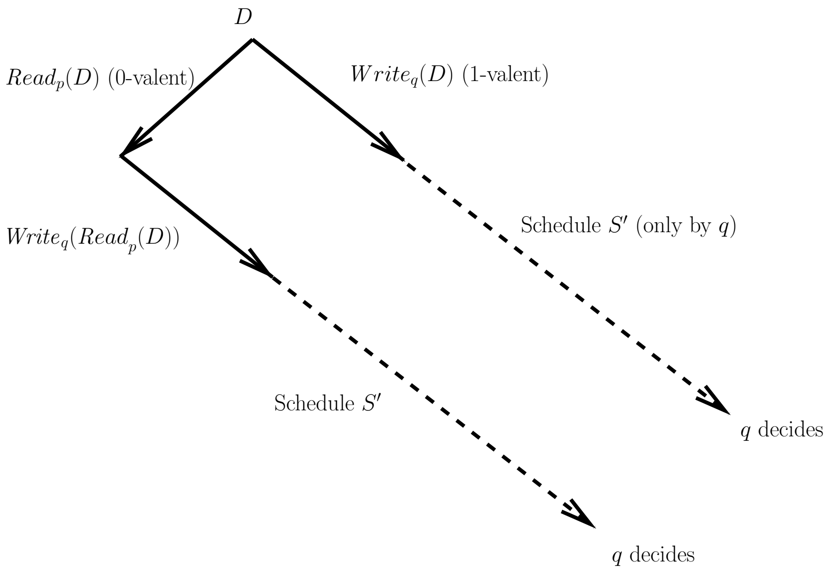


Figure 11.2.: Read and write issued on the same register

  Recall that $q(p(D)$ is 0-valent and $q(D)$ is 1-valent. Take any sufficiently long schedule S only containing steps of q, such that some process q decides in $S(q(p(D)))$. Since q cannot distinguish $S(q(p(D)))$ from $S(q(D))$, it should decide the same value in $S(q(D))$.

  But $q(p(D))$ is 0-valent and $p(D)$ is 1-valent—a contradiction.

  The case when p writes in its next step in D is symmetric.

$\square_{\textit{Theorem } 29}$

As solving consensus for one process is trivial, the following result is immediate from Theorem 29.

Corollary 5 $cons(\textsf{atomic-register}) = 1$

11.4. Objects with consensus numbers 2

In this section, we show that the hierarchy of object types based on consensus numbers is "populated": for ever n, there exists an object type T, such that $cons(T) = n$. We begin with showing that objects types test&set and queue have consensus number 2.

11.4.1. Consensus from test&set objects

A test&set object stores a binary value, initially 0, and exports a single (atomic) *test&set* operation that writes 1 to it and returns the old value. Its sequential specification is defined as follows:

```
operation $X.test\&set$ ():
    $loc := X$;
    $X := 1$;
    return ($prev$).
```

Thus, the first process to access a (non-initialized) test&set object hets 0 (we call it a *winner*) and all subsequent processes get 1.

The consensus algorithm described in Figure 11.3 uses one test&set object TS and two 1W1R atomic registers $REG[0]$ and $REG[1]$.

When the process p_i (for convenience, we assume that $i \in \{0, 1\}$) invokes $propose(v)$ on the consensus object, it "publishers" its input value v in $REG[i]$ (line 1).

Then p_i accesses TS (line 2). If it wins, it decides its own input value (line 3). Otherwise, it decides the value proposed by the other process p_{1-i} (line 4). Intuitively, as exactly one process wins TS, only the value proposed by the winner can be decided.

```
operation $propose(v)$ issued by $p_i$:
(1)    $REG[i] := v$;
(2)    $aux := TS.test\&set$ ();
(3)    if ($aux = 0$) then return ($v$)
(4)       ($aux = 1$) else return ($REG[1 - i]$)
```

Figure 11.3.: From test&set to consensus

Theorem 30 *The algorithm in Figure 11.3 is a wait-free consensus implementation for two processes using test&set objects and atomic registers.*

Proof As every process performs at most three shared-memory steps before deciding, the algorithm is clearly wait-free.

Let p_i be the process that, in a given execution of the algorithm, accesses TS first and decides its own input value v. By the algorithm, p_i previously wrote v in atomic register $REG[i]$. Thus, p_{1-i} that accesses TS after p_i, will after that find v in $REG[i]$ and return it.

Thus, the two processes can only return that inout value of the winner, and the agreement and validity properties of consensus are satisfied.. $\square_{Theorem\ 30}$

11.4.2. Consensus from queue objects

Recall that a queue object exports two operations $enqueue$ and $dequeue$, where $enqueue(v)$ adds element v to the end of the queue and $dequeue()$ removes the element at the head of the queue and returns it; if the queue is empty, the default value $\perp$ is returned.

A wait-free consensus algorithm for two processes that uses two registers and a queue is presented in Figure 11.4. The algorithm assume that the queue is initialized with the sequence of items $< w, \ell >$. The first process first to perform a dequeue operation on this queue gets w and considers itself a winner. As in the previous algorithm, the value proposed by the winner will be decided.

Using the arguments of the proof of Theorem 30, we obtain:

```
operation $propose(v)$ issued by $p_i$:
(1)     $REG[i] := v$;
(2)     $aux := Q.dequeue()$;
(3)     if $(aux = w)$ then return $(REG[i])$
(4)        $(aux = \ell)$ else return $(REG[1-i])$
```

Figure 11.4.: From queue to consensus

Theorem 31 *The algorithm in Figure 11.4 is a wait-free consensus implementation for two processes using* queue *objects and atomic registers.*

11.4.3. Consensus numbers of test&set and queue

As we have shown, test&set and queue objects , combined with atomic registers, can be used to wait-free implement consensus in a system of two processes. We show below that the objects have consensus number 2, i.e., they cannot be used to solve consensus for three or more processes.

Theorem 32 *There does not exist a wait-free consensus implementation for three processes from objects of types in* {test&set, queue, atomic-registers}.

Proof By contradiction, suppose that there exists a wait-free consensus algorithm A for two processes, p, q, and r using atomic registers, test&set objects and queues.

By Lemma 29, A has a critical configuration D, i.e., D is bivalent, $p(D)$, $q(D)$, and $r(D)$ are monovalent, and all the three processes are about to access the same object X. Without loss of generality, assume that $p(D)$ is 0-valent, while $q(D)$ and $r(D)$ are 1-valent.

It is immediate from the proof of Theorem 29 that X must be a test&set object or a queue.

1. X is a test&set object.

   The two *test&set* operations on X performed by p and q result in two configurations $q(p(D))$ and $p(q(D))$ that only p and q can distinguish: the state of r and the states of all objects (including X) are identical in the two configurations.

   Consider a schedule S in which r runs solo (neither p nor q appear in S) starting from $q(p(D))$ and r decides in $S(q(p(D)))$. Since $p(D)$ is 0-valent, r must decide 0 in $S(q(p(D)))$. But $S(q(p(D))$ is indistinguishable to r from $S(p(q(D)))$—a contradiction with the assumption that $q(D)$ is 1-valent.

2. X is a queue.

   Let OP_p the operation issued by p that leads from D to $p(D)$, OP_q the operation issued by q that leads from D to $q(D)$, and OP_r the operation issued by r that leads from D to $r(D)$.

   Here we consider the following possible subcases:

   - OP_p and OP_q are *dequeue* operations.

     Then, regardless of the state of X in D, $q(p(D))$ and $p(q(D))$ are identical, except for the local states of P and q. Thus, in a solo schedule, r can never distinguish two configurations of opposite valences—a contradiction.

- $\textsc{op}_p$ is an *enqueue* operation and $\textsc{op}_q$ is a *dequeue* operation.

  If, in configuration D, X is empty, then $q(p(D))$ and $q(D)$ only differ in the local states of p and q, and X is left empty in both configurations.

  If X is non-empty in D, then $q(p(D))$ and $p(q(D))$ are identical.

  In both cases, in solo extensions, r cannot distinguish two configurations of opposite valences—a contradiction.

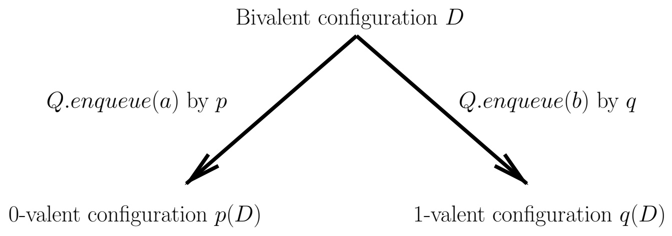

Bivalent configuration D
$Q.enqueue(a)$ by p
$Q.enqueue(b)$ by q
0-valent configuration $p(D)$
1-valent configuration $q(D)$

Figure 11.5.: $enqueue()$ operations by p and q

- Now we are left with the most interesting case: $\textsc{op}_p$ and $\textsc{op}_q$ are *enqueue* operations, let a and b be, respectively, the arguments of the two operations.

  Configurations $q(p(D))$ and $p(q(D))$ differ only in the state of X: in $q(p(D))$, the element enqueued by p precedes the element enqueued by q, and in $q(p(D))$—vice versa.

  Consider a solo schedule of p applied to $q(p(D))$. To decide, p must be able to distinguish the run from a run starting applied $q(p(D))$, p should eventually access X.

  Let S_p be the solo schedule of p such that in $S_p(q(p(D)))$, p is *about* to dequeue element a it previously enqueued (in operation $\textsc{op}_p$).

  Note that in $S_p(q(p(D)))$ and $S_p(p(q(D)))$ differ only in the state of X and, thus, to decide in a solo schedule applied to $S_p(q(p(D)))$, q must eventually access X to dequeue its own element in X enqueued by operation $\textsc{op}_q$.

  Similarly, Let S_q be the solo schedule of p such that in $S_q(S_p(q(p(D))))$, q is *about* to dequeue element b it previously enqueued (in operation $\textsc{op}_p$).

  Finally, we observe that $S_q(S_p(q(p(D))))$ and $S_q(S_p(p(q(D))))$ still differ only in the state of X (Figure 11.5): in the first configuration, X begins with $a; b$ and in the second configuration—with $a; b$. Thus, by the dequeue operations of p and q in reversed orders, we obtain two identical configurations, $q(p(S_q(S_p(q(p(D))))))$ and $p(q(S_q(S_p(p(q(D))))))$, of opposite valences—a contradiction.

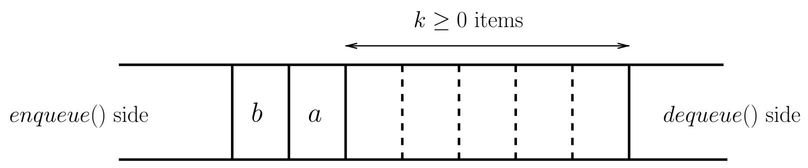


Figure 11.6.: State of the queue object Q in configuration $q(p(D))$

□*Theorem 32*

Theorems 30, 31, and 32 imply

Corollary 6 $cons(\textsf{test\&set}) = cons(\textsf{queue}) = 2$.

```
operation $propose(v)$:
    $\ell := \ell + 1$
    if $(x = \perp)$ then $x := v$
    if $(\ell \leq n)$ then
        return $(x)$;
    else
        return $(\perp)$;
```

Figure 11.7.: Consensus specification: sequential execution of $popose(v)$

11.5. Objects of n-consensus type

In this section, we show that for each $n \in \mathbb{N}$, there exists object types T, such that $cons(T) = n$, i.e., the hierarchy of object types implied by their consensus numbers is populated for each level n.

The sequential specification of the n-consensus object type is given in Figure 11.7. The state of an n-consensus object is defined by two variables: x (initially $\perp$)—the value to be decided and ℓ (initially 0)—the number of *propose* operations perfomed on the object so far. As with the consensus type, the argument of the first *propose* operation fixes x. However, only first n *propose* operation return a decided value. All subsequent operations return $\perp$.

We suggest the reader to compute the consensus number of the type, following the lines of the proofs above:

Theorem 33 *For all $n \in \mathbb{N}$, $cons(n\text{-consensus}) = n$.*

11.6. Objects whose consensus number is $+\infty$

We now complete the picture by showing that some object types have an infinite consensus number: atomic objects of these types, combined with atomic registers can be used to solve consensus among any number of processes. We discuss two such object types: compare&swap objects and augmented queue.

11.6.1. Consensus from compare&swap objects

A compare&swap object that stores a *value* x exports a single $compare\&swap()$ operation that takes two values as arguments, *old* and *new*, with the following sequential specification:

```
operation $compare\&swap(old, new)$:
    $prev := x$;
    if $(x = old)$ then $x := new$;
    return $(prev)$
```

From compare&swap objects to consensus Implementing consensus from a single compare&swap object in a system of any number n of processes is straightforward (Figure 11.8) The base $compare\&swap$ object CS is initialized to $\perp$, a default value that cannot be proposed to the consensus object. When a process proposes a value v, it invokes $CS.compare\&swap(\perp, v)$ (line 1). If $\perp$ is returned, the process decides its value (line 2). Otherwise, it decides the value returned by the compare&swap object (line 3).

```
operation $propose(v)$ issued by $p_i$:
(1)    $aux := CS.compare\&swap(\perp, v)$;
(2)    if $aux = \perp$ then $return(v)$
(3)    else $return(aux)$
```

Figure 11.8.: From compare&swap to consensus

Theorem 34 $cons($*compare&swap*$) = \infty$.

Proof The algorithm in Figure 11.8 is clearly wait-free. Let p_i be the first process to execute $CS.compare\&swap\ ()$ operation in a given execution. (Recall that "the first" is defined based on the linearization order on operations on CS.) Clearly, any subsequent call of $CS.compare\&swap\ ()$ returns the input value of p_i and, thus, only this value can be decided. $\Box_{Theorem\ 34}$

11.6.2. Consensus from augmented queue objects

An augmented-queue object is a previously considered queue with an additional $peek()$ operation that returns the first item of the queue without removing it. Intuitively, the object type has infinite consensus power, as the first element to be enqueued can then be "peeked" and returned as a decision value (assuming that the queue is initially empty).

```
operation $propose(v)$ issued by $p_i$:
        $Q.enqueue(v)$;
        $return(Q.peek())$
```

Figure 11.9.: From an augmented queue to consensus

Figure 11.9 gives a simple wait-free implementation of a consensus object from an augmented queue. The construction is pretty simple. The augmented queue Q is initially empty. A process first enqueues its input value and then invokes the $peek()$ operation to obtain the first value that has been enqueued. It is easy to see that the construction works for any number of processes, and we have the following theorem:

Theorem 35 $cons($*augmented-queue*$) = \infty$.

11.7. Consensus hierarchy

Consensus numbers establish a hierarchy on the power of object types to wait-free implement a consensus object, i.e., to wait-free implement any object defined by a sequential specification on total operations. As we have shown, the lowest level object types (of consensus number 1) include atomic-registers, the second weakest class of object types (of consensus number 2) includes test&set and queue, and the strongest class (of consensus number ∞) includes compare&swap and augmented-queue. We also showed that for all $n \in \mathbb{N}$, there are object types, e.g., n-consensus, that have consensus number exactly n, i.e., every level in the hierarchy is "populated."

Consensus numbers also allow ranking the power of classical synchronization primitives (provided by shared memory parallel machines) in presence of process crashes: $compare\&swap$ is stronger than $test\&set$ that is, in turn, stronger than atomic read/write operations. Interestingly, they also show that classical objects encountered in sequential computing such as stacks and queues are as powerful as the test&set or fetch&add synchronization primitives when one is interested in providing upper layer application processes with wait-free objects.

Fault-tolerance can be impossible to achieve when the designer is not provided with powerful enough atomic synchronization operations. As an example, a FIFO queue that has to tolerate the crash of a single process, cannot be built from atomic registers. This follows from the fact that the consensus number of a queue is 2, while the he consensus number of atomic registers is 1.

Bibliographic notes

The hierarchy of object types based on consensus numbers was originally introduced by Herlihy [35]. The article also contains multiple examples of how the consensus number of an object type can be computed. Jayanti observed that the consensus hierarchy, as defined originally by Herlihy, is not *robust*: there are combinations of lower level types that turn out to be stronger than a higher level type [41]. To fix this, Jayanti proposes a refined definition that has been used since then. The question of robustness of the resulting consensus hierarchy remains however open. Lo and Hadzilacos [54] give examples of *non-deterministic* types that give a higher level type under composition, but it remains unclear whether deterministic types are robust.

The impossibility of implementing wait-free consensus for two processes using atomic registers presented in this chapter involves elements (valence and critical configurations) of the original proof by Fisher, Lynch and Paterson [27] who showed that even 1-*resilient* (i.e., tolerating the failure of a single process) consensus is impossible to solve in an asynchronous message-passing system. Loui and Abu-Amara extended the proof to read-write shared-memory systems [55].

In this book, we get the 1-resilient consensus impossibility (Chapter **??**) by a simulation-based reduction to the wait-free impossibility.

Exercises

1. Complete the proof of Lemma 29 by confirming that if a configuration D satisfies the first two properties of a critical configuration, but not the third one, then there exist descendants $p(D)$ and $q(D)$ such that $p(D)$ is 0-valent and $q(D)$ is 1-valent.

2. Prove Corollary 6.

Part V.

Schedulers

Bibliography

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

[2] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *PODC*, pages 159–170, 1993.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, Oct. 1985.

[4] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, page 483485, 1967.

[5] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 237–246, 1996.

[6] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the 28th Symposium on Foundations of Computer Science*, pages 337–346. IEEE Computer Society Press, Oct. 1987.

[7] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[8] H. Attiya, A. Fouren, and E. Gafni. A polynomial adaptive algorithm for long-lived $(2k-1)$-renaming. Technical report, 2003. Unpublished manuscript, private communication.

[9] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 122–136, 2005.

[10] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, 2004.

[11] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 249–259, 1987.

[12] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, 1993.

[13] H. P. Brinch, editor. *The Origin of Concurrent Programming*. Springer Verlag, 2002. 534 pages.

[14] J. E. Burns and G. L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 222–231, 1987.

[15] H. C.A.R. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[16] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[17] S. Chaudhuri, M. Kosa, and J. Welch. One-write algorithms for multivalued regular and atomic register. *Acta Informatica*, 37(161-192), 2000.

[18] S. Chaudhuri and J. L. Welch. Bounds on the costs of multivalued register implementations. *SIAM J. Comput.*, 23(2):335–354, 1994.

[19] O.-J. Dahl, E. Dijkstra, and H. C.A.R. *Structured Programming*. Academic Press, 1972. 220 pages.

[20] C. Delporte-Gallet, H. Fauconnier, E. Gafni, and L. Lamport. Adaptive register allocation with a linear number of registers. In *International Symposium on Distributed Computing*, DISC '13, pages 269–283, 2013.

[21] C. Delporte-Gallet, H. Fauconnier, E. Gafni, and S. Rajsbaum. Linear space bootstrap communication schemes. *Theoretical Computer Science*, 561:122–133, 2015.

[22] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8, 1965.

[23] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997.

[24] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *J. ACM*, 46(5):633–666, Sept. 1999.

[25] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, Sept. 1998.

[26] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of the International Symposium on Distributed Computing*, pages 493–494, 2005.

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[28] F. C. Freiling, R. Guerraoui, and P. Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 2011.

[29] E. Gafni and S. Rajsbaum. Distributed programming with tasks. In *OPODIS*, pages 205–218, 2010.

[30] R. Guerraoui, M. Kapałka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 399–412, 2006.

[31] R. Guerraoui and E. Ruppert. Linearizability is not always a safety property. In *Networked Systems - Second International Conference, NETYS 2014*, pages 57–69, 2014.

[32] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University, May 1994.

[33] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186–203, Jan. 1995.

[34] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, Jan. 1991.

[35] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.

[36] M. Herlihy, D. N. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

[37] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[38] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[39] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[40] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[41] P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.

[42] P. Jayanti, J. Burns, and G. Peterson. Almost optimal single reader single writer atomic register. *Journal of Parallel and Distributed Computing*, 60:150–168, 2000.

[43] P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[44] D. N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(1):1–13, 2012.

[45] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[46] L. Lamport. Proving the correctness of multiprocessor programs. *Transactions on software engineering*, 3(2):125–143, Mar. 1977.

[47] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, Sept. 1979.

[48] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Prog. Lang. Syst.*, 6(2):254–280, Apr. 1984.

[49] L. Lamport. On interprocess communication; part I: Basic formalism; part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986.

[50] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst.*, 4(3):382–401, July 1982.

[51] M. Li, J. Tromp, and P. Vityani. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, 1996.

[52] N. Linial. Doing the IIS. Unpublished manuscript, 2010.

[53] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE1:7–19, 1975.

[54] W. Lo and V. Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000.

[55] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[56] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[57] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):143–153, 1986.

[58] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[59] D. Parnas. On the criteria to be used in decomposing systems in to module. *Communications of the ACM*, 15(2):1053–1058–336, 1972.

[60] D. Parnas. A technique for software modules with examples. *Communications of the ACM*, 15(2):330–336, 1972.

[61] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[62] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[63] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986.

[64] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[65] A. K. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):331–334, 1994.

[66] G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Prentice-Hall, 2006.

[67] J. Tromp. How to construct an atomic variable (extended abstract). In *WDAG*, pages 292–302, 1989.

[68] J. Tromp. *Aspects of Algorithms and Complexity*. PhD thesis, Universiteit van Amsterdam, 1993.

[69] K. Vidyasankar. Converting Lamport's regular register to atomic register. *Information Processing Letters*, 28(6):287–290, 1988.

[70] K. Vidyasankar. An elegant 1-writer multireader multivalued atomic register. *Information Processing Letters*, 30(5):221–223, 1989.

[71] K. Vidyasankar. A very simple cosntruction of 1-writer multireader multivalued atomic variable. *Information Processing Letters*, 37:323–326, 1991.

[72] P. M. B. Vitányi. Simple wait-free multireader registers. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 118–132, 2002.

[73] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 233–243, 1986.

[74] W. E. Weihl. Atomic data types. *IEEE Database Eng. Bull.*, 8(2):26–33, 1985.