Chapter 6
Distributed Shared Memory

© Ajay Kshemkalyani, 2004. Use in CS 566 at UIC, Spring 2006.

6.1 Abstraction and Advantages

Distributed shared memory (DSM) is an abstraction provided to the programmer of a distributed system. It gives the impression of a single monolithic memory, as in a traditional von Neumann architecture. The programmer accesses the data across the network using only *read* and *write* primitives, as he would in a uniprocessor system. The programmer does not have to deal with *send* and *receive* communication primitives and the ensuing problems of dealing with synchronization and consistency. The DSM abstraction is illustrated in Figure 1. A part of each computer's memory is earmarked for shared space, and the remainder is private memory. To provide the programmers the illusion of a single shared address space, a memory mapping management layer is required to manage the *shared virtual memory* space.

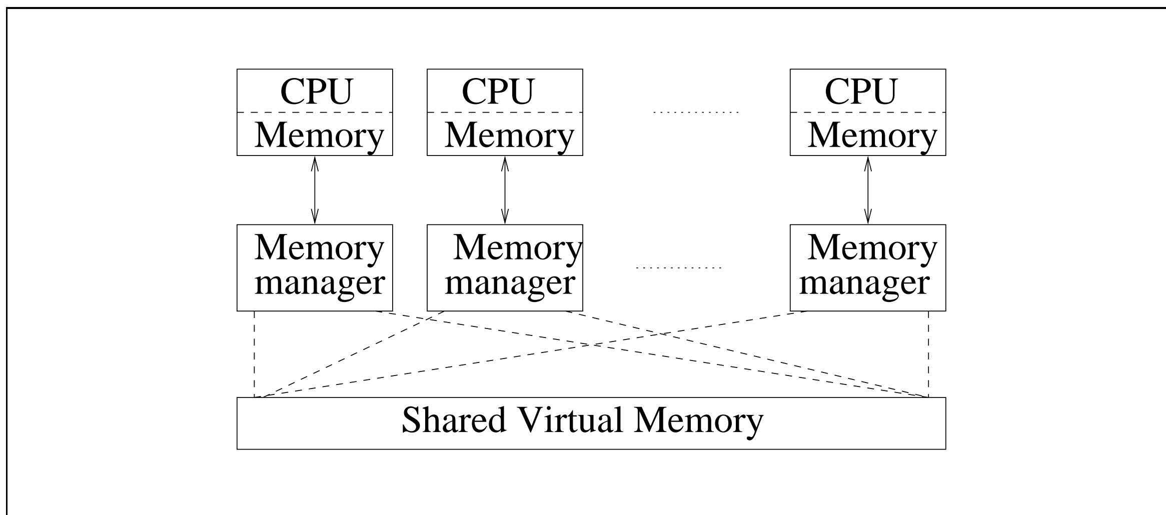

Figure 1: Abstract view of DSM

The following are the advantages of DSM.

1. Communication across the network is achieved by the read/write abstraction that simplifies the task of the programmer.

2. A single address space is provided, thereby providing the possibility of avoiding data movement across multiple address spaces by using *passing-by-reference*, instead of *passing-by-value*.

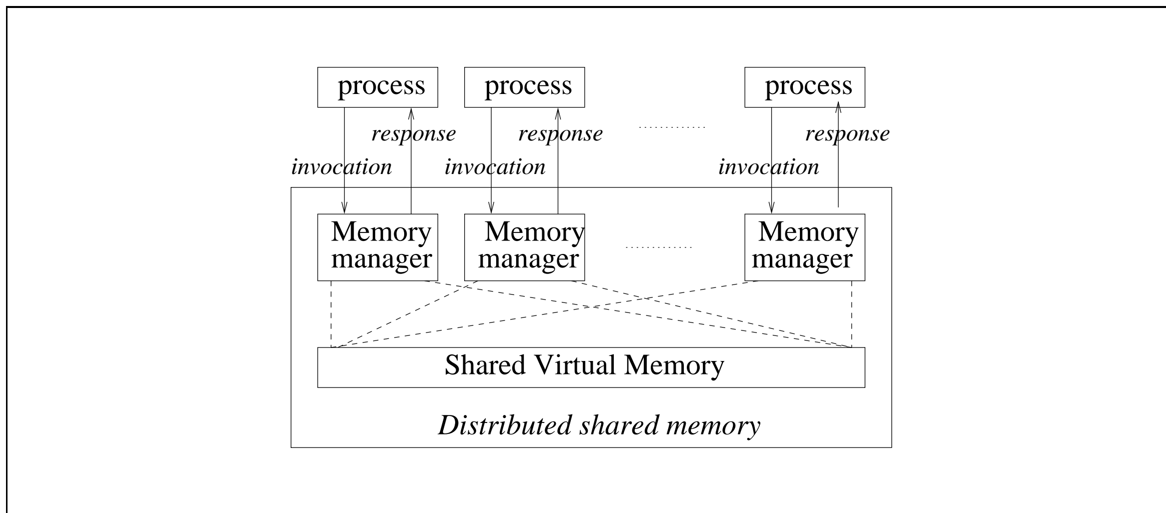

Figure 2: Detailed abstraction of DSM and interaction with application processes.

3. If a block of data needs to be moved, the system can exploit locality of reference to reduce the communication overhead.

4. DSM is often cheaper than using dedicated multiprocessor systems, because it uses simpler software interfaces and off-the-shelf hardware.

5. There is no bottleneck presented by a single memory access bus.

6. DSM effectively provides a large (virtual) main memory.

7. DSM provides portability of programs written using DSM. This portability arises due to a common DSM programming interface, that is independent of the operating system and other low-level system characteristics.

Although a familiar (i.e., read/write) interface is provided to the programmer, there is a catch to it. Under the covers, there is inherently a distributed system and a network, and the data needs to be shared in some fashion. There is no silver bullet. Moreover, with the possibility of data replication and/or the concurrent access to data, concurrency control needs to be enforced. Specifically, when multiple processors wish to access the same data object, a decision about how to handle concurrent accesses needs to be made. As in traditional databases, if a locking mechanism based on read and write locks for objects is used, concurrency is severely restrained, defeating one of the purposes of having the distributed system. On the other hand, if concurrent access is permitted by different processors to different replicas, the problem of replica consistency (which is a generalization of the problem of cache consistency in computer architecture studies) needs to be addressed. The main point of allowing concurrent access (by different processors) to the same data object is to increase throughput. But in the face of concurrent access, the semantics of what value a read operation returns to the program needs to be specified. The programmer ultimately needs to understand this semantics, which may differ from the Von Neumann semantics, because the program logic depends greatly on this semantics. This compromises the assumption that the DSM is transparent to the programmer.

Before examining the challenges in implementing replica coherency in DSM systems, we look at the disadvantages.

 © A. Kshemkalyani, 2004

1. The programmer is not shielded from having to know about various replica consistency models and coding his distributed application according to the semantics of these models.

2. As DSM is implemented under the covers using asynchronous message-passing, the overheads incurred are at least as high as those of a message-passing implementation. As such, DSM implementations cannot be more efficient than asynchronous message-passing implementations. The generality of the DSM software may make it less efficient.

3. By yielding control to the DSM memory management layer, the programmer loses the ability to use his own message-passing solutions for accessing shared objects. It is likely that the standard vanilla implementations of DSM have a higher overhead than a programmer-written implementation tailored for a specific application and system.

The main issues in designing a DSM system are the following.

- Determining what semantics to allow for concurrent access to shared objects. The semantics needs to be clearly specified so that the programmer can code his program using an appropriate logic.

- Determining the best way to implement the semantics of concurrent access to shared data. One possibility is to use replication. One decision to be made is the degree of replication – partial replication at some sites, or full replication at all the sites. A further decision then is to decide on whether to use read-replication (replication for the read operations) or write-replication (replication for the write operations) or both.

- Selecting the locations for replication (if full replication is not used), to optimize efficiency from the system's viewpoint.

- Determining the location of remote data that the application needs to access, if full replication is not used.

- Reducing communication delays and the number of messages that are involved under the covers while implementing the semantics of concurrent access to shared data.

There is a wide range of choices on how these issues can be addressed. In part, the solution depends on the system architecture. Recall from Chapter 1 that DSM systems can range from tightly-coupled (hardware and software) multicomputers to wide-area distributed systems with heterogenous hardware and software. There are four broad dimensions alng which DSM systems can be classified and implemented.

- Whether data is replicated or cached
- Whether remote access is by hardware or by software
- Whether the caching/replication is controlled by hardware or software
- Whether the DSM is controlled by the distributed memory managers, by the operating system, or by the language runtime system.

The various options for each of these four dimensions, and their comparison, are shown in Figure 1.

Type of DSM	Examples	Management	Caching	Remote access
single-bus multiprocessor	Firefly, Sequent	by MMU	hardware control	by hardware
switched multiprocessor	Alewife, Dash	by MMU	hardware control	by hardware
NUMA system	Butterfly, CM*	by OS	software control	by hardware
Page-based DSM	Ivy, Mirage	by OS	software control	by software
Shared variable DSM	Midway, Munin	by language runtime system	software control	by software
Shared object DSM	Linda, Orca	by language runtime system	software control	by software

Table 1: Comparison of DSM systems

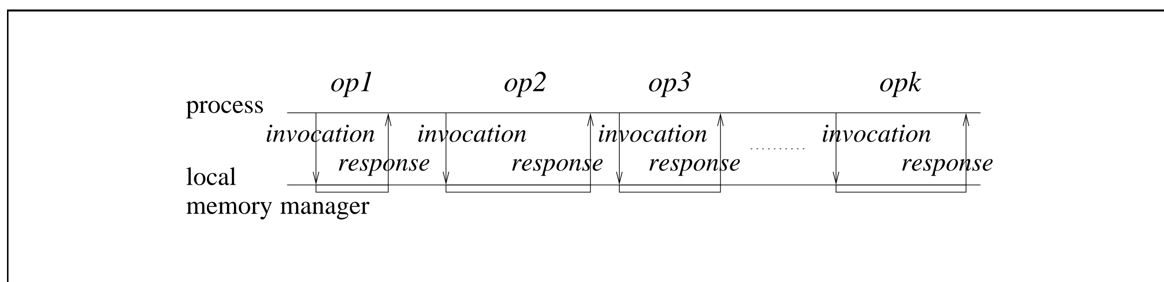

Figure 3: Sequential invocations and responses in a DSM system, without any pipelining.

6.2 Memory Consistency Models

Memory coherence is the ability of the system to execute memory operations correctly. Assume n processes and s_i memory operations per process P_i. Also assume that all the operations issued by a process are executed sequentially (that is, pipelining is disallowed), as shown in Figure 3. Observe that there are a total of

$$(s_1 + s_2 + \ldots + s_n)!/(s_1!s_2! \ldots s_n!)$$

possible permutations or interleavings of the operations issued by the processes. The problem of ensuring memory coherence then becomes the problem of identifying which of these interleavings are "correct", which of course requires a clear definition of "correctness". The *memory consistency model* defines the set of allowable memory access orderings. While a traditional definition of correctness says that a correct memory execution is one that returns to each *Read* operation, the value stored by the most recent *Write* operation, the very definition of "most recent" becomes ambigious in the presence of multiple replicas of the data item. Thus, a clear definition of correctness is required in such a system; the objective is to disallow the interleavings that make no semantic sense, while not being overly restrictive so as to permit a high degree of concurrency.

The DSM system enforces a particular memory consistency model; the programmer writes his program keeping in mind the allowable interleavings permitted by that specific memory consistency model. A program written for one model may not work correctly on a DSM system that enforces a different model. The model can thus be viewed as a *contract* between the DSM system and the programmer using that system. We now consider six consistency models, that are related as shown in Figure 12.

Notation: A write of value a to variable x is denoted as *Write(x,a)*. A read of variable x that returns value a is denoted as *Read(x,a)*. A subscript on these operations is sometimes used to denote the processor that issues these operations.

6.2.1 Strict consistency/Atomic consistency/Linearizability

The strictest model, corresponding to the notion of correctness on the traditional Von Neumann architecture or the uniprocessor machine, requires that any *Read* to a location (variable) should return the value written by the most recent *Write* to that location (variable). Two salient features of such a system are the following. (i) A common global time axis is implicitly available in a uniprocessor system. (ii) Each write is immediately visible to all processes. Adapting this correctness model to a DSM system with operations that can be concurrently issued by the various processes gives the *strict consistency model*, also known as the *atomic consistency model*, that is more formally specified as follows.

1. Any *Read* to a location (variable) is required to return the value written by the most recent *Write* to that location (variable) as per a global time reference.

   For operations that do not overlap as per the global time reference, the specification is clear. For operations that overlap as per the global time reference, the following further specifications are necessary.

2. All operations appear to be executed atomically and sequentially.

3. All processors see the same ordering of events, which is equivalent to the global-time occurrence of non-overlapping events.

An equivalent way of specifying this consistency model is in terms of the 'invocation' and 'response' to each *Read* and *Write* operation. Recall that each operation takes a finite time interval and hence different operations by different processors can overlap in time. However, the invocation and the response to each invocation can both be separately viewed as being atomic events. An execution sequence in global time is viewed as a sequence Seq of such invocations and responses. Clearly, Seq must satisfy the conditions:

- (Liveness:) Each invocation must have a corresponding response, and
- (Correctness:) The projection of Seq on any processor i, denoted Seq_i, must be a sequence of alternating invocations and responses if pipelining is disallowed.

Despite the concurrent operations, a linearizable execution needs to generate an equivalent global order on the events, that is a permutation of Seq, satisfying the semantics of *linearizability.*

More formally, a sequence Seq of invocations and responses is *linearizable* (LIN) if there is a permutation Seq' of adjacent pairs of corresponding $\langle invoc, resp \rangle$ events satisfying:

1. For every variable v, the projection of Seq' on v, denoted Seq'_v, is such that

   - every *Read* (adjacent $\langle invoc, resp \rangle$ event pair) returns the most recent *Write* (adjacent $\langle invoc, resp \rangle$ event pair) that immediately preceded it.

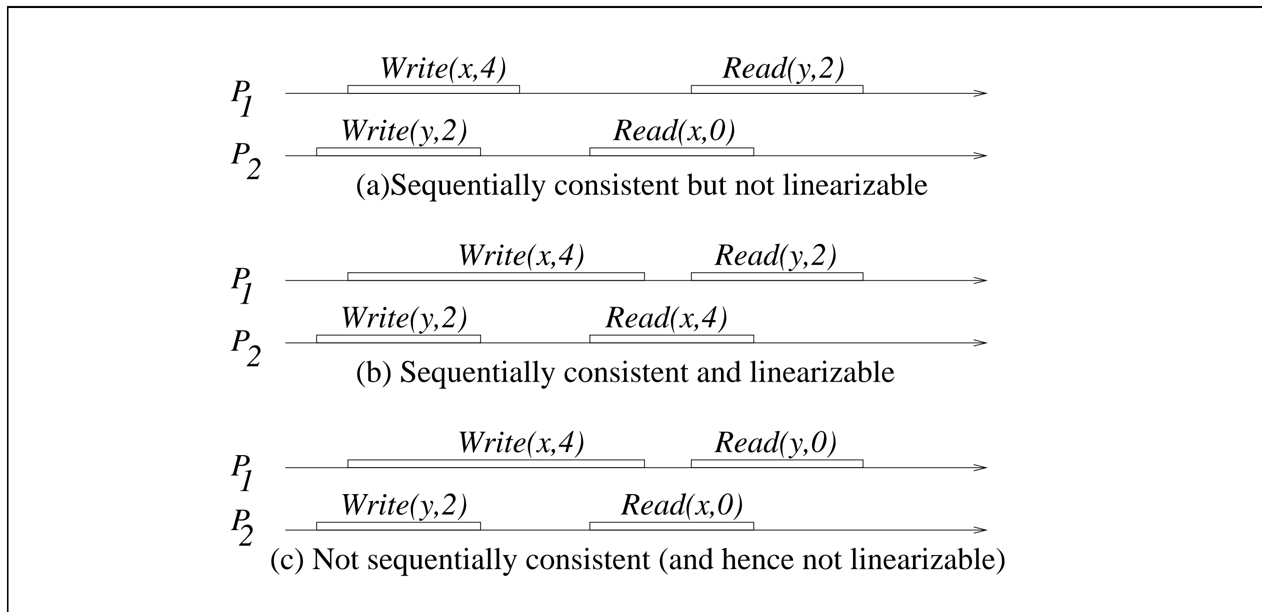

Figure 4: Examples to illustrate definitions of linearizability and sequential consistency. The initial values of variables are zero.

2. If the response $op1(resp)$ of operation $op1$ occurred before the invocation $op2(invoc)$ of operation $op2$ in Seq, then $op1$ (adjacent $\langle invoc, resp\rangle$ event pair) occurs before $op2$ (adjacent $\langle invoc, resp\rangle$ event pair) in Seq'.

Condition 1 specifies that every processor sees a common order Seq' of events, and that in this order, the semantics is that each *Read* returns the most recent completed *Write* value. Condition 2 specifies that the common order Seq' must satisfy the global time order of events, viz., the order of non-overlapping operations in Seq must be preserved in Seq'.
Examples: Figure 4 shows three executions.

Figure 4(a): The execution is not linearizable because although the *Read* by P_2 begins after *Write*$(x, 4)$, the *Read* returns the value that existed before the *Write*. Hence, a permutation Seq' satisfying the above condition(2) on global time order does not exist.

Figure 4(b): The execution is linearizable. The global order of operations (corresponding to $\langle$response, invocation$\rangle$ pairs in Seq'), consistent with the real-time occurrence is: *Write*$(y, 2)$, *Write*$(x, 4)$, *Read*$(x, 4)$, *Read*$(y, 2)$. This permutation Seq' satisfies the conditions (1 and 2).

Figure 4(c): The execution is not linearizable. The two dependencies: *Read*$(x, 0)$ before *Write*$(x, 4)$, and *Read*$(y, 0)$ before *Write*$(x, 2)$ cannot both be satisfied in a global order while satisfying the local order of operations at each processor. Hence, there does not exist any permutation Seq' satisfying conditions (1 and 2).

Implementations

Implementing linearizability is expensive because a global time scale needs to be simulated. As all processors need to agree on a common order, the implementation needs to use total or-

```
(shared var)
int: x;

(1) When the Memory Manager receives a Read or Write from application:
(1a) total_order_broadcast the Read or Write request to all processors;
(1b) await own request that was broadcast;
(1c) perform pending response to the application as follows
(1d)        case Read: return value from local replica;
(1e)        case Write: write to local replica and return ack to application.

(2) When the Memory Manager receives a total_order_broadcast(Write, x, val) from network:
(2a) write val to local replica of x.

(3) When the Memory Manager receives a total_order_broadcast(Read, x) from network:
(3a) no operation.
```

Figure 5: Implementing Linearizability (LIN) using total order broadcasts. Code shown is for P_i, $1 \leq i \leq n$.

der. For simplicity, we assume full replication of each data item at all the processors. Hence, total ordering needs to be combined with a broadcast. Figure 5 gives the implementation assuming the existence of a *total order broadcast* primitive that broadcasts to all processors including the sender. Hence, the Memory Manager software has to be placed between the application above it and the total order broadcast layer below it.

Although the algorithm in Figure 5 appears simple, it is also subtle. The total order broadcast ensures that all processors see the same order.

- For two nonoverlapping operations at different processors, by the very definition of nonoverlapping, the response to the former operation precedes the invocation of the latter in global time.
- For two overlapping operations, the total order ensures a common view by all processors.

For a *Read* operation, when the Memory Managers systemwide receive the total order broadcast, they do not perform any action. Why is the broadcast then necessary? The reason is this. If *Read* operations do not participate in the total order broadcasts, they do not get totally ordered with respect to the *Write* operations as well as with respect to the other *Read* operations. This can lead to a violation of linearizability, as shown in Figure 6.2.1. The *Read* by P_k returns the value written by P_i. The later *Read* by P_i returns the initial value of 0. As per the global time ordering requirement of linearizability, the *Read* by P_j that occurs after the *Read* by P_k must also return the value 4. However, that is not the case in this example, wherein the *Read* operations do not participate in the total order broadcast.

6.2.2 Sequential Consistency

Linearizability or strict/atomic consistency is difficult to implement because the absence of a global time reference in a distributed system necessitates that the time reference has to be

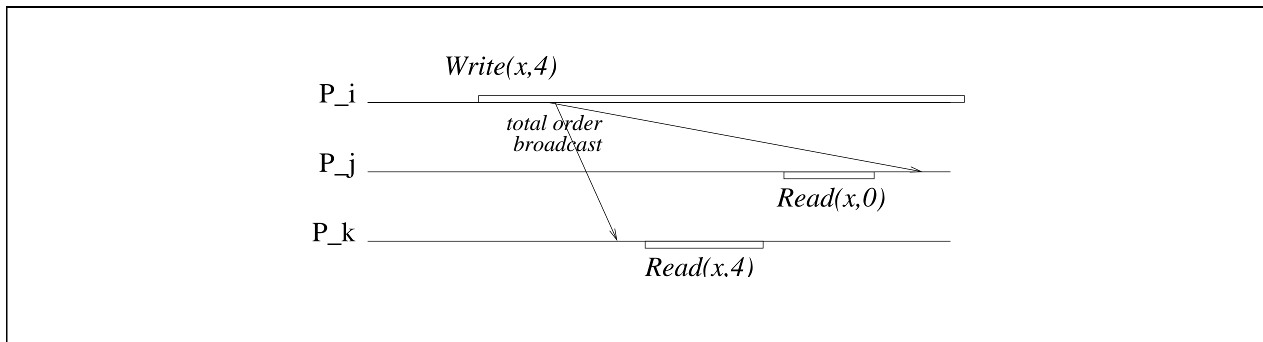

Figure 6: A violation of *linearizability* (LIN) if *Read* operations do not participate in the total order broadcast.

simulated, which is very expensive. Programmers can deal with weaker models. The first weaker model, that of *sequential consistency* (SC) was proposed by Lamport and uses logical time reference instead of the global time reference.

Sequential consistency is specified as follows.

- The result of any execution is the same as if all operations of the processors were executed in *some* sequential order.
- The operations of each individual processor appear in this sequence in the local program order.

Although any possible interleaving of the operations from the different processors is possible, all the processors must see *the same* interleaving. In this model, even if two operations from different processors (on the same or different variables) do not overlap in a global time scale, they may appear in reverse order in the *common* sequential order seen by all the processors.

More formally, a sequence Seq of invocation and response events is sequentially consistent if there is a permutation Seq' of adjacent pairs of corresponding $\langle invoc, resp \rangle$ events satisfying:

1. For every variable v, the projection of Seq' on v, denoted Seq'_v, is such that:
   - every *Read* (adjacent $\langle invoc, resp \rangle$ event pair) returns the most recent *Write* (adjacent $\langle invoc, resp \rangle$ event pair) that immediately preceded it.
2. If the response $op1(resp)$ of operation $op1$ at process P_i occurred before the invocation $op2(invoc)$ of operation $op2$ by process P_i in Seq, then $op1$ (adjacent $\langle invoc, resp \rangle$ event pair) occurs before $op2$ (adjacent $\langle invoc, resp \rangle$ event pair) in Seq'.

Condition (1) is the same as that for linearizability. Condition (2) differs from that for linearizability. It specifies that the common order Seq' must satisfy only the local order of events at each processor, instead of the global order of nonoverlapping events. Hence the order of non-overlapping operations issued by different processors in Seq need not be preserved in Seq'.

Examples: Three examples are considered in Figure 4.

```
(shared var)
int: x;

(1) When the Memory Manager at P_i receives a Read or Write from application:
(1d) case Read: return value from local replica;
(1e) case Write(x,val): total_order_broadcast_i(Write(x,val) to all processors including itself.

(2) When the Memory Manager at P_i receives a total_order_broadcast_j(Write, x, val) from network:
(2a) write val to local replica of x;
(2b) if i = j then return acknowledgement to application.
```

Figure 7: Implementing Sequential Consistency (SC) using local *Read* operations. Code shown is for P_i, $1 \leq i \leq n$.

Figure 4(a): The execution is sequentially consistent: the global order Seq' is: $Write(y, 2)$, $Read(x, 0)$, $Write(x, 4)$, $Read(y, 2)$.

Figure 4(b): As the execution is linearizable (seen in Section 6.2.1), it is also sequentially consistent. The global order of operations (corresponding to ⟨response, invocation⟩ pairs in Seq'), consistent with the real-time occurrence is: $Write(y, 2)$, $Write(x, 4)$, $Read(x, 4)$, $Read(y, 2)$.

Figure 4(c): The execution is not sequentially consistent (and hence not linearizable). The two dependencies: $Read(x, 0)$ before $Write(x, 4)$, and $Read(y, 0)$ before $Write(x, 2)$ cannot both be satisfied in a global order while satisfying the local order of operations at each processor. Hence, there does not exist any permutation Seq' satisfying conditions (1 and 2).

Implementations

As sequential consistency (SC) is less restrictive than linearizability (LIN), it should be easier to implement it. As all processors are required to see the same global order, but global time orderng need not be preserved across processes, it is sufficient to use total order broadcasts for the *Write* operations only. In the simplified algorithm, no total order broadcast is required for *Read* operations, because:

1. all consecutive operations by the same processor are ordered in that same order because of not using pipelining, and
2. *Read* operations by different processors are independent of each other and need to be ordered only with respect to the *Write* operations in the execution.

In Exercise 1, you will be asked to reason this more thoroughly. Two algorithms for SC are next given, that exhibit a trade-off of the inhibition of *Read* versus *Write* operations.

Local-Read algorithm: The first algorithm for SC, given in Figure 7, is a direct simplification of the algorithm for linearizability, given in Figure 5. In the algorithm, a *Read*

operation completes atomically, whereas a *Write* operation does not. Between the invocation of a *Write* by P_i (line 1e) and its acknowledgement (lines 2a,2b), there may be multiple *Write* operations initiated by other processors that take effect at P_i (line 2a). Thus, a *Write* issued locally has its completion locally delayed. Such an algorithm is acceptable for *Read*-intensive programs.

Local-Write algorithm: The algorithm in Figure 8 does not delay acknowledgement of *Write*s. For *Write*-intensive programs, it is desirable that a locally issued *Write* gets acknowledged immediately (as in lines 2a-2c), even though the total order broadcast for the *Write*, and the actual update for the *Write* may not go into effect by updating the variable at the same time (line 3a). The algorithm achieves this at the cost of delaying a *Read* operation by a processor until all previously issued local *Write* operations by that same processor have locally gone into effect (i.e., previous *Write*s issued locally have updated their local variables being written to). The variable *counter* is used to track the number of *Write* operations that have been locally initiated but not completed at any time. A *Read* operation completes only if there are no prior locally initiated *Write* operations that have not written to their variables (line 1a), i.e., there are no pending locally initiated *Write* operations to any variable. Otherwise, a *Read* operation is delayed until after all previously initiated *Write* operations have written to their local variables (lines 3b-3d), which happens after the total order broadcasts associated with the *Write* have delivered the broadcast message locally.

This algorithm performs fast (local) *Write*s and slow *Read*s. The algorithm pipelines all *Write* updates issued by a processor. The *Read* operations have to wait for all *Write* updates issued earlier by that processor to complete (i.e., take effect) locally before the value to be read is returned to the application.

6.2.3 Causal Consistency

Under sequential consistency, it is required that *Write* operations issued by different processors must necessarily be seen in some common order by all processors. This requirement can be relaxed to require only that *Write*s that are *causally related* must be seen in that same order by all processors, whereas 'concurrent' *Writes* may be seen by different processors in different orders. The resulting consistency model is the *causal consistency* model. We have seen the definition of causal relationships among events in a message passing system. What does it mean for two *Write* operations to be causally related?

The *causality relation* for shared memory systems is defined as follows.

Local order: At a processor, the serial order of the events defines the local causal order

Inter-process order: A *Write* operation causally precedes a *Read* operation issued by another processor if the *Read* returns a value written by the *Write*.

Transitive closure: The transitive closure of the above two relations defines the (global) causal order.

Examples: The examples in Figure 9 illustrate causal consistency.

(shared var)
int: x;

(1) <u>When the Memory Manager at P_i receives a *Read*(x) from application:</u>
(1a) **if** *counter* $= 0$ **then**
(1b) **return** x.

(2) <u>When the Memory Manager at P_i receives a *Write(x,val)* from application:</u>
(2a) *counter* $\longleftarrow$ *counter* $+ 1$;
(2b) **total_order_broadcast**$_i$ the *Write*(x, val);
(2c) **return** acknowledgement to the application.

(3) <u>When the Memory Manager at P_i receives a **total_order_broadcast**$_j$(*Write, x, val*) from network:</u>
(3a) **write** *val* to local replica of x.
(3b) **if** $i = j$ **then**
(3c) *counter* $\longleftarrow$ *counter* $- 1$;
(3d) **if** (*counter* $= 0$ and any *Read*s are pending) **then**
(3e) **perform** pending responses for the *Read*s to the application.

Figure 8: Implementing Sequential Consistency (SC) using local *Write* operations. Code shown is for P_i, $1 \leq i \leq n$.

Figure 9(a): The execution is sequentially consistent (and hence causally consistent). Both P_3 and P_4 see the operations at P_1 and P_2 in sequential order and in causal order.

Figure 9(b): The execution is not sequentially consistent but it is causally consistent. Both P_3 and P_4 see the operations at P_1 and P_2 in causal order because the lack of a causality relation between the *Write*s by P_1 and by P_2 allows the values written by the two processors to be seen in different orders in the system. The execution is not sequentially consistent because there is no global satisfying the contradictory ordering requirements set by the *Read*s by P_3 and by P_4. What can be said if the two *Read* operations of P_4 returned 7 first and then 4? (See Exercise 4.)

Figure 9(c): The execution is not causally consistent because the second *Read* by P_4 returns 4 after P_4 has already returned 7 is an earlier *Read*.

Implementation

We first examine the definition of sequential consistency. Even though all processors only need to see *some* total order of the *Write* operations, observe that if two *Write* operations are related by causality (i.e., the second *Write* begins causally after a *Read* that reads the value written by the first *Write*), then the order of the two *Writes* seen by all the processors also satisfies causal order! In the implementation, even though a total-order-broadcast primitive is used, observe that it implicitly provides causal ordering on all the *Write* operations. Thus, due to the nature of the definition of causal ordering in shared memory systems, a total-order-broadcast also provides causal order broadcast, unlike the case for message-passing systems. (Exactly why is it so?)

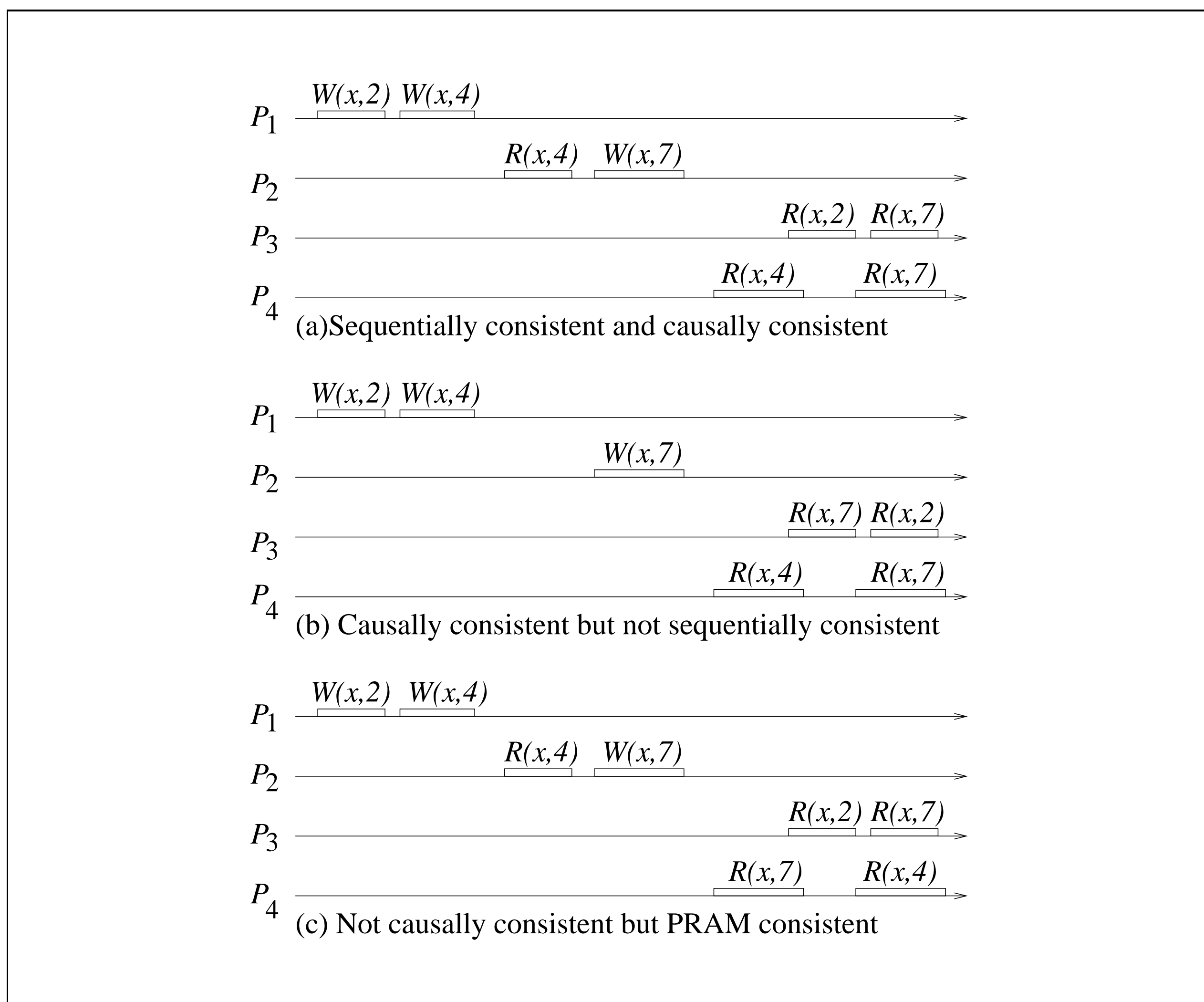

Figure 9: Examples to illustrate definitions of sequential consistency (SC), causal consistency (CC), and PRAM consistency. The initial values of variables are zero.

In contrast to the SC requirement, causal consistency implicitly requires only that causal order be provided. Thus, a causal-order-broadcast can be used in the implementation. The details of the implementation are left as Exercise 5.

6.2.4 PRAM (Pipelined RAM) or Processor Consistency

Causal consistency requires all causally-related *Writes* to be seen in the same order by all processors. This may be viewed as being too restrictive for some applications. A weaker form of consistency requires only that *Write* operations issued by the same (any one) processor are seen by all other processors in that same order in which they were issued, but *Write* operations issued by different processors nay be seen in differing orders by different processors. In relation to the 'causality' relation between operations, only the local causality relation, as defined by the local order of *Write* operations, needs to be seen by other processors. Hence, this form of consistency is termed *processor consistency.* An equivalent name for this consistency model is *Pipelined RAM* (PRAM), to capture the behavior that all operations issued by any processor appear to the other processors in a FIFO pipelined

```
(shared variables)
int: x, y;

Process i                              Process j

...                                    ...
(1a) x ←— 4;                           (2a) y ←— 6;
(1b) if y = 0 then kill(P2).           (2b) if x = 0 then kill(P1).
```

Figure 10: A counter-intuitive behaviour of a PRAM-consistent program. The initial values of variables are zero.

sequence.
Examples:

- In Figure 9(c), the execution is PRAM consistent (even though it is not causally consistent) because (trivially) both P_3 and P_4 see the updates made by P_1 and P_2 in FIFO order along the channels P_1 to P_3 and P_2 to P_3, and along P_1 to P_4 and P_2 to P_4, respectively.
- While PRAM consistency is more permissive than causal consistency, this model must be used with care by the programmer because it can lead to rather unintuitive results. For example, examine the code in Figure 10, where x and y are shared variables. It is possible that on a PRAM system, both processes P_1 and P_2 get killed. This can happen as follows. (i) P_1 writes 4 to x in line (1a) and P_2 writes 6 to y in line (2a) at about the same time. (ii) Before these written values propagate to the other processor, P_1 reads y (as being 0) in line (1b) and P_2 reads x (as being 0) in line (2b). Here, a *Read* (e.g., in (1b) or (2b)) can effectively 'overtake' a preceding *Write* (of (2a) or (1a), resp.) if the two accesses by the same processor are to different locations. However, this would not be expected on a conventional machine, where at most one process may get killed, depending on the interleaving of the statements.
- The execution in Figure 11(a) violates PRAM consistency. An explanation is given in Section 6.2.5.

Implementations

PRAM consistency can be implemented using FIFO broadcast. The implementation details are left an Exercise 6.

6.2.5 Slow Memory

The next weaker consistency model is that of *slow memory*. This model represents a location-relative weakening of the PRAM model. In this model, only all *Write* operations issued by the same processor and to the same memory location must be observed in the same order by all the processors.
Examples: The examples in Figure 11 illustrate slow memory consistency.

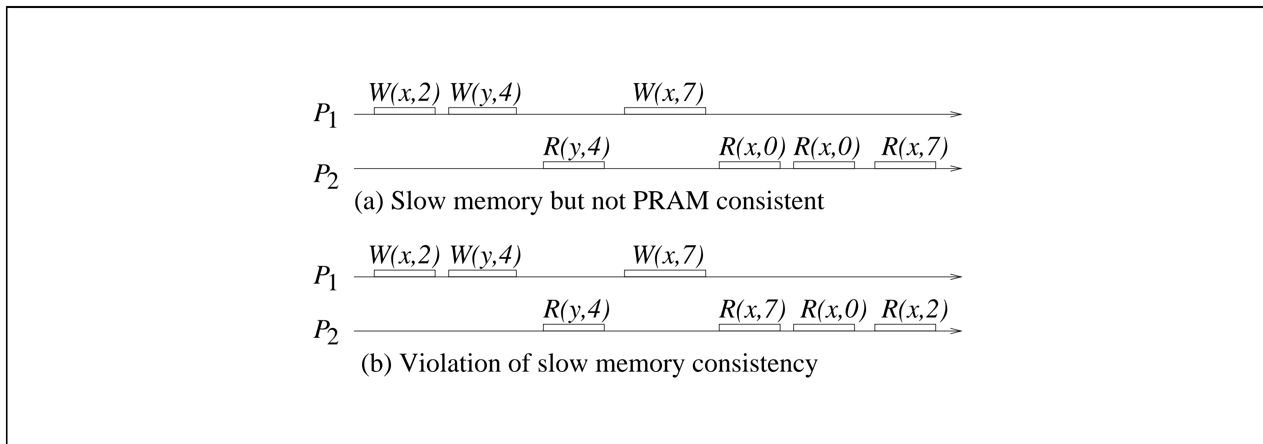

Figure 11: Examples to illustrate definitions of PRAM consistency and slow memory. The initial values of variables are zero.

Figure 11(a): The updates to reach of the variables are seen pipelined separately in a FIFO fashion. The 'x' pipeline from P_1 to P_2 is slower than the 'y' pipeline from P_1 to P_2. Thus, the overtaking effect is allowed. However, PRAM consistency is violated because the FIFO property is violated over the single common 'pipeline' from P_1 to P_2 – the update to y is seen by P_2 but the much older value of $x = 0$ is seen by P_2 later.

Figure 11(b): Slow memory consistency is violated because the FIFO property is violated for the pipeline for variable x. '$x = 7$' is seen by P_2 before it sees '$x = 0$' and '$x = 2$' although 7 was written to x after the values of 0 and 2.

Implementations

Slow memory can be implemented using a broadcast primitive that is weaker than even the FIFO broadcast. What is required is a FIFO broadcast per variable in the system, i.e., the FIFO property should be satisfied only for updates to the same variable. The implementation details are left as Exercise 7.

6.2.6 Hierarchy of Consistency Models

Based on the definitions of the memory consistency models seen so far, there exists a hierarchy among the models, as depicted in Figure 12.

6.2.7 Other Models based on Synchronization Instructions

We have seen several popular consistency models. Based on the consistency model, the behaviour of the DSM differs, and the programmer's logic therefore depends on the underlying consistency model. It is also possible that newer consistency models may arise in the future.

All the consistency models seen so far apply to all the instructions in the distributed program. We now briefly mention some other consistency models that are based on a different principle, namely that the consistency conditions apply only to a set of distinguished 'synchronization' or 'coordination' instructions. These synchronization instructions are typically

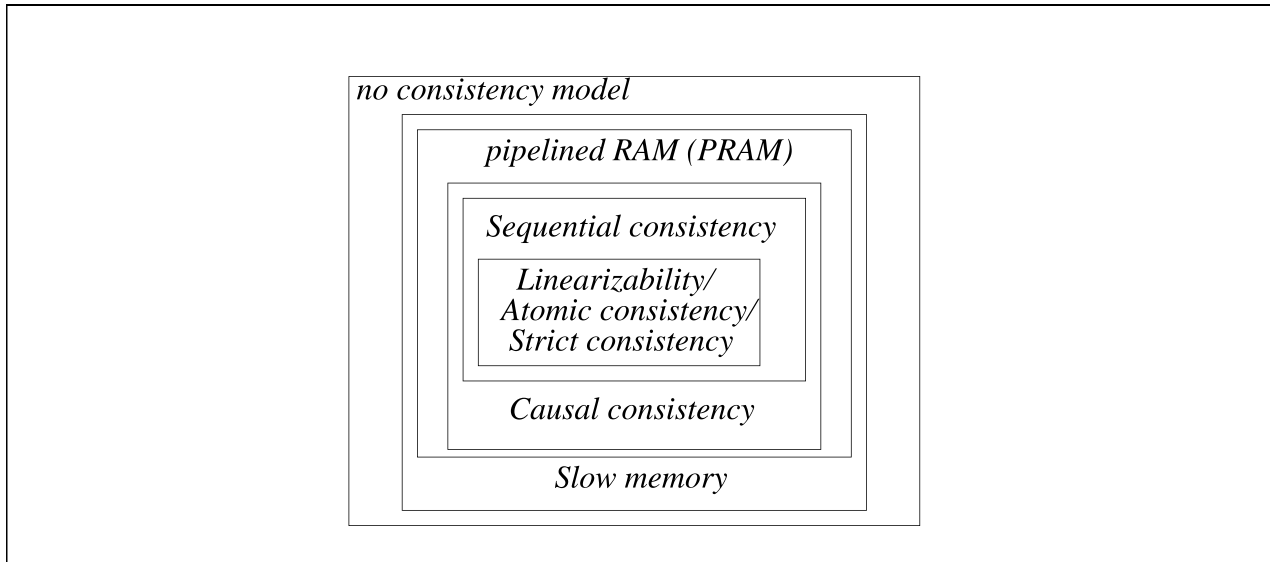

Figure 12: A strict hierarchy of the memory consistency models.

from some run-time library. A common example of such a statement is the barrier synchronization. Only the synchronization statements across the various processors must satisfy the consistency conditions; other program statements between synchronization statements may be executed by the different processors without any conditions. Examples of consistency models based on this principle are: *entry consistency*, *weak consistency*, and *release consistency*. The synchronization statements are inserted in the program based on the semantics of the types of accesses. For example, accesses may be conflicting (to the same variable) or non-conflicting (to different variables), conflicting accesses may be competing (a *Read* and a *Write*, or two *Write*s) or non-conflicting (two *Read*s), and so on. We outline the definitions of these consistency models but skip further implementation details of such models.

Weak Consistency:
Some applications do not require even seeing all writes, let alone seeing them in some order. Consider the case of a process executing a CS, repeatedly reading and writing some variables in a loop. Other processes are not supposed to read or write these variables until the first process has exited its CS. However, if the memory has no way of knowing when a process is in a CS and when it is not, the DSM has to propagate all writes to all memories in the usual way. But by using synchronization variables, processes can deduce whether the CS is occupied.

A synchronization variable in this model has the following *semantics*: it is used to propagate all writes to other processors, *and* to perform local updates with regard to changes to global data that occurred elsewhere in the distributed system. When synchronization occurs, all *Writes* are propagated to other processes, and all *Write*s done by others are brought locally. In an implementation specifically for the CS problem, updates can be propagated in the system only when the synchronization variable is accessed (indicating an entry or exit into the CS).

Weak consistency has the following three properties which guarantee that memory is consistent at the synchronization points.

1. Accesses to synchronization variables are sequentially consistent.

2. No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.

3. No data access (either *Read* or *Write*) is allowed to be performed until all previous accesses to synchronization variables have been performed.

An access to the synchronization variable forces *Write* operations to complete, and effectively flushes the pipelines. Before reading shared data, a process can perform synchronization to ensure it accesses the most recent data.

Release Consistency:
The drawback of *weak consistency* is that when a synchronization variable is accessed, the memory does not know whether this is being done because the process is finished writing the shared variables (exiting the CS) or about to begin reading them (entering the CS). Hence, it must take the actions required in both cases.

- Ensuring that all locally initiated *Writes* have been completed, i.e., propagated to all other processes.
- Ensuring that all *Writes* from other machines have been locally reflected.

If the memory could differentiate between entering the CS and leaving the CS, a more efficient implementation is possible. To provide this information, two kinds of synchronization variables or operations are needed instead of one.

Release consistency provides these two kinds. *Acquire* accesses are used to tell the memory system that a critical region is about to be entered. Hence, the actions for Case (2) above need to be performed to ensure that local replicas of variables are made consistent with remote ones. *Release* accesses say that a critical region has just been exited. Hence, the actions for Case (1) above need to be performed to ensure that remote replicas of variables are made consistent with the local ones that have been updated. The *Acquire* and *Release* operations can be defined to apply to a subset of the variables. The accesses themselves can be implemented either as ordinary operations on special variables or as special operations.

If the semantics of a CS is not associated with the *Acquire* and *Release* operations, then the operations effectively provide for *barrier synchronization.* Until all processes complete the previous phase, none can enter the next phase.

The following rules are followed by the protected variables in the general case.

- All previously initiated *Acquire* operations must complete successfully before a process can access a protected shared variable.
- All accesses to a protected shared variable must complete before a *Release* operation can be performed.
- The *Acquire* and *Release* operations effectively follow the PRAM consistency model.

A relaxation of the release consistency model is called the *lazy release consistency* model. Rather than propagating the updated values throughout the system as soon as a process leaves a critical region (or enters next phase in the case of barrier synchronization), the updated values are propagated to the rest of the system only on demand, i.e., only when

they are needed. Changes to shared data are only communicated when an *Acquire* access is performed by another process.

Entry Consistency:
Entry consistency requires the programmer to use *Acquire* and *Release* at the start and at the end of each CS, respectively. But unlike release consistency, entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier. When an *Acquire* is performed on a synchronization variable, only access to those ordinary shared variables that are guarded by that synchronization variable is regulated.

6.3 Shared Memory Mutual Exclusion

Operating systems have traditionally dealt with multi-process synchronization using algorithms based on first principles (e.g., the well-known bakery algorithm), high-level constructs such as *semaphores* and *monitors*, and special 'atomically executed' instructions supported by special-purpose hardware (e.g., *Test-&-Set*, *Swap*, and *Compare-&-Swap*). These algorithms are applicable to all shared memory systems. In this section, we will review the bakery algorithm which requires $O(n)$ accesses in the entry section, irrespective of the level of contention. We will then study *fast mutual exclusion* which requires $O(1)$ accesses in the entry section in the absence of contention. This algorithm also illustrates an interesting technique in resolving concurrency. As hardware primitives have the in-built atomicity that helps to easily solve the mutual exclusion problem, we will then examine mutual exclusion based on these primitives.

6.3.1 Lamport's Bakery Algorithm

Lamport proposed the classical *bakery algorithm* for n-process mutual exclusion in shared memory systems. The algorithm is so called because it mimics the actions that customers follow in a bakery store. A process wanting to enter the critical section picks a token number that is one greater than the elements in the array $choosing[1 \dots n]$. Processes enter the critical section in the increasing order of the token numbers. In case of concurrent accesses to *choosing* by multiple processes, the processes may have the same token number. In this case, a unique *lexicographic order* is defined on the tuple $\langle token, pid \rangle$, and this dictates the order in which processes enter the critical section. The algorithm for process i is given in Figure 13. The algorithm can be shown to satisfy the three requirements of the critical section problem: (i) mutual exclusion, (ii) bounded waiting, and (iii) progress.

In the entry section, a process chooses a timestamp for itself, and resets it to 0 in the exit section. In steps (1a)-(1c), each process chooses a timestamp for itself, as the max of the latest timestamps of all processes, plus one. These steps are non-atomic; thus multiple processes could be choosing timestamps in overlapping durations. When process i reaches (1d), it has to check the status of each other process j, to deal with the effects of any race conditions in selecting timestamps. In (1d)-(1f), process i serially checks the status of each other process j. If j is selecting a timestamp for itself, j's selection interval may have overlapped with that of i, leading to an unknown order of timestamp values. Process i needs to make sure that any other process j ($j < i$) that had begin to execute (1b) concurrently with itself and may still be executing (1b) does not assign itself the same timestamp. Otherwise mutual exclusion could be violated as i would enter the CS, and subsequently, j, having a

 © A. Kshemkalyani, 2004

```
(shared vars)
array of boolean: choosing[1...n];
array of integer: timestamp[1...n];

repeat
(1) P_i executes the following for the entry section:
(1a) choosing[i] <-- 1;
(1b) timestamp[i] <-- max_{k∈[1...n]}(timestamp[k]) + 1;
(1c) choosing[i] <-- 0;
(1d) for count = 1 to n do
(1e)       while choosing[count] do no-op;
(1f)       while timestamp[count] ≠ 0 and (timestamp[count], count) < (timestamp[i], i) do
(1g)             no-op.

(2) P_i executes the critical section (CS) after the entry section

(3) P_i executes the following exit section after the CS:
(3a) timestamp[i] <-- 0.

(4) P_i executes the remainder section after the exit section

until false;
```

Figure 13: Lamport's n-process bakery algorithm for shared memory mutual exclusion. Code shown is for process Pi, $1 \leq i \leq n$.

lower process identifier and hence a lexicographically lower timestamp, would also enter the CS. Hence, i waits for j's timestamp to stabilize, i.e., *choosing*$[j]$ to be set to *false*. Once j's timestamp is stabilized, i moves from (1e) to (1f). Either j is not requesting (in which case j's timestamp is 0) or j is requesting. Step (1f) determines the relative priority between i and j. The process with a *lexicographically* lower timestamp has higher priority and enters the CS; the other process has to wait (step (1g)). Hence, *mutual exclusion* is satisfied.

Bounded waiting is satisfied because each other process j can "overtake" process i at most once after i has completed choosing its timestamp. The second time j chooses a timestamp, the value will necessarily be larger than i's timestamp if i has not yet entered its CS. *Progress* is guaranteed because the lexicographic order is a total order and the process with the lowest timestamp at any time in the loop (1d)-(1g) is guaranteed to enter the CS.

Attempts to improve the bakery algorithm have lead to several important results.

- *Space complexity:* A lower bound of n registers, specifically, the *timestamp* array, has been shown for the shared memory critical section problem. Thus, one cannot hope to have a more space-efficient algorithm for distributed shared memory mutual exclusion.

- *Time complexity:* In many environments, the level of contention may be low. The $O(n)$ overhead of the entry section does not scale well for such environments. This concern is addressed by the field of *fast mutual exclusion* that aims to have $O(1)$ time overhead for the entry and exit sections of the algorithm, in the absence of contention.

Although this algorithm guarantees mutual exclusion and progress, unfortunately, this fast algorithm has a price – in the worst case, it does not guarantee bounded delay. Next, we will study Lamport's algorithm for fast mutual exclusion in asynchronous shared memory systems. This algorithm is notable in that it is the first algorithm for fast mutual exclusion, and uses the asynchronous shared memory model. Further, it illustrates an important technique for resolving contention. The worst-case unbounded delay in the presence of persisting contention has been addressed subsequently, by using a timed model of execution, wherein there is an upper bound on the time it takes to execute any step. We will not discuss mutual exclusion under the timed model of execution.

6.3.2 Lamport's RWRWR Mechanism and Fast Mutual Exclusion

Lamport's *fast mutual exclusion* algorithm is given in Figure 14. The algorithm illustrates an important technique – the $\langle W - R - W - R\rangle$ sequence that is a necessary and sufficient sequence of operations to check for contention and to ensure safety in the entry section, using only two registers.

```
start :    1:   b[i] ⟵ true;
           2:   x ⟵ i;
           3:   if y ≠ 0 then
           4:                    b[i] ⟵ false;
           5:                    await y = 0;
           6:                    goto start fi;
           7:   y ⟵ i;
           8:   if x ≠ i then
           9:                    b[i] ⟵ false;
          10:                    for j = 1 to N   do await ¬b[j] od;
          11:                    if y ≠ i then
          12:                                     await y = 0;
          13:                                     goto start fi fi;
        [CS]:   critical section;
          14:   y ⟵ 0;
          15:   b[i] ⟵ false;
```

Figure 14: Lamport's deadlock-free solution for process P_i, where $1 \leq i \leq n$, using $\Omega(n)$ registers.

Steps (2), (3), (7) , and (8) represent a basic $\langle W(x) - R(y) - W(y) - R(x)\rangle$ sequence whose necessity in identifying a minimal sequence of operations for fast mutual exclusion is justified as follows.

1. The first operation needs to be a *Write*, say to variable x. If it were a *Read*, then all contending processes could find the value of the variable even outside the entry section.

2. The second operation cannot be a *Write* to another variable, for that could equally be combined with the first *Write* to a larger variable. The second operation should *not* be a *Read* of x because it follows *Write* of x and if there is no interleaved operation from another process, the *Read* does not provide any new information. So the second operation must be a *Read* of another variable, say y.

3. The sequence must also contain *Read(x)* and *Write(y)* because there is no point in reading a variable that is not written to, a writing a variable that is never read.

4. The last operation in the minimal sequence of the entry section must be a *Read*, as it will help determine whether the process can enter CS. So the last operation should be *Read(x)*, and the second-last operation should be the *Write(y)*.

In the absence of contention, each process writes its own id to x and then reads y. Then finding that y has its initial value, the process writes its own id to y and then reads x. Finding x to still be its own id, it enters CS. Correctness needs to be shown in the presence of contention – let us discuss this after considering the structure of the remaining entry and exit section code.

In the exit section, the process must do a Write to indicate its completion of the CS. The Write cannot be to x which is also the first variable written in the entry section. So the operation must be Write(y).

Now consider the following sequence of interleaved operations by processes i, j, and k in the entry section.

$$W_j(x)\langle x=j, y=0\rangle, W_i(x)\langle \mathbf{x}=\mathbf{i}, y=0\rangle, R_i(y)\langle x=i, y=0\rangle, R_j(y)\langle x=i, y=0\rangle, W_i(y)\langle x=i, \mathbf{y}=\mathbf{i}\rangle,$$

$$W_j(y)\langle x=i, y=j\rangle, R_i(x)\langle x=i, y=j\rangle, W_k(x)\langle x=k, y=j\rangle, R_j(x)\langle x=k, y=j\rangle,$$

Process i enters its critical section, but there is no record of its identity or that it had written any variables at all, because the variables it wrote (shown boldfaced above) have been overwritten. In order that other processes can discover when (and who) leaves the CS, there needs to be another variable that is set before the CS and reset after the CS. This is the boolean, $flag[i]$. Additionally, y needs to be reset on exiting the CS.

The code in lines (3)-(6) has the following use. If a process p finds $y \neq 0$, then another process has executed at least line (7) and not yet executed line (14). So process p resets its own flag, and before retrying again, it awaits for $y = 0$. If process p finds $y = 0$ in line (3), it sets $y = p$ in line (7) and checks if $x = p$.

- If $x = p$, then no other process has executed line (2), and any later process would be blocked in the line (3)-(6) loop now because $y = p$. Thus, if $x = p$, process p can safely enter the CS.

- If $x \neq p$, then another process, say q, has overwritten x in line (2) and there is a potential race. Two broad cases are possible.

  - Process q finds $y \neq 0$ in line (3). It resets its flag, and stays in the line (3)-(6) section at least until p has exited the CS. Process p on the other hand resets its own flag (line (9)) and waits for all other process such as q to reset their own flags. As process q is trapped in lines (3)-(6), process p will find $y = i$ in line (11) and enter the CS.

```
(shared variables among the processes accessing each of the different object types)
register: Reg ⟵ initial value;                      // shared register initialized
(local variables)
integer: old ⟵ initial value;                       // value to be returned

(1) Test&Set(Reg) returns value:
(1a) old ⟵ Reg;
(1b) Reg ⟵ 1;
(1c) return(old).

(2) Swap(Reg, new) returns value:
(2a) old ⟵ Reg;
(2b) Reg ⟵ new;
(2c) return(old).
```

Figure 15: Definitions of synchronization operations *Test&Set* and *Swap*.

- Process q finds $y = 0$ in line (3). It sets y to q in line (7), and enters the race, even closer to process p which is at line (8). Of the processes such as p and q that contend at line (8), there will be a unique winner.
  - If no other process r has since written to x in line (2), the winner is the process among p and q that executed line (2) last, i.e., wrote its own id to x. That winner will enter the CS directly from line (8), whereas the losers will reset their own flags, await the winner to exit and reset its flag, and also await other contenders at line (8) and newer contenders to reset their own flags. The losers will compete again from line (1) after the winner has reset y.
  - If some other process r has since written its id to x in line (2), both p and q will enter code in lines (9)-(13). Both p and q reset their flags, await for r which will be trapped in lines (4)-(6) to reset its flag, and then both p and q check the value of y. Between p and q, the process that last wrote to y in line (7) will become the unique winner and enter the CS directly. The loser will then await for the winner to reset y, and then compete again from line (1).

Thus, mutual exclusion is guaranteed, and progress is also guaranteed. However, a process may be starved, although with decreasing probability, as its number of attempts increases.

6.3.3 Hardware support for mutual exclusion

Hardware support can allow for special instructions that perform two or more operations atomically. Two such instructions, *Test&Set* and *Swap*, are defined and implemented as shown in Figure 15. The atomic execution of two actions (a *Read* and a *Write* operation) can greatly simplify a mutual excluison algorithm, as seen from the mutual exclusion code in Figure 16 and Figure 17, respectively. The algorithm in Figure 16 can lead to starvation. The algorithm in Figure 17 is enhanced to guarantee bounded waiting by using a "round-robin" policy to selectively grant permission when releasing the critical section.

```
(shared variables)
register: Reg ⟵ false;                              // shared register initialized
(local variables)
integer: blocked ⟵ 0;                               // value to be returned

repeat
(1) P_i executes the following for the entry section:
(1a) blocked ⟵ true;
(1b) repeat
(1c)      Swap(Reg, blocked);
(1d) until blocked = false;

(2) P_i executes the critical section (CS) after the entry section

(3) P_i executes the following exit section after the CS:
(3a) Reg ⟵ false;

(4) P_i executes the remainder section after the exit section

until false;
```

Figure 16: Mutual exclusion using *Swap*. Code shown is for process Pi, $1 \leq i \leq n$.

6.4 Wait-freedom

Processes that interact with each other, whether by message passing or by shared memory, need to synchronize their interactions. Traditional solutions to synchronize asynchronous processes via shared memory objects (also called *concurrent objects*) use solutions based on locking, busy waiting, critical sections, semaphores, or conditional waiting. An arbitrary delay of a process or its crash failure can prevent other processes from completing their operations. This is undesirable.

Wait-freedom is a property that guarantees that any process can complete any synchronization operation in a finite number of lower-level steps, irrespective of the execution speed of other processes. More precisely, a wait-free implementation of a concurrent object guarantees that any process can complete an operation on it in a finite number of steps, irrespective of whether other processes crash or encounter unexpected delays. Thus, processes that crash, or encounter unexpected delays (such as delays due to high processor load, swapping out of memory, or CPU schedulng policies) should not delay other processes in a wait-free implementation of a concurrent object.

Not all synchronizations have wait-free solutions. As a trivial example, a producer-consumer synchronization between two processes cannot be implemented in a wait-free manner if the producer process crashes before posting its value – the consumer is necessarily blocked. Nevertheless, the notion of wait-freedom is an important concept in designing fault-tolerant systems and algorithms whenever possible. An alternate view of wait-freedom in terms of fault-tolerance is as follows.

- An f-resilient system is a system in which up to f of the n processes can fail, and

```
(shared variables)
register: Reg ⟵ false;                          // shared register initialized
array of boolean: waiting[1...n];
(local variables)
integer: blocked ⟵ initial value;               // value to be returned

repeat
(1) P_i executes the following for the entry section:
(1a) waiting[i] ⟵ true;
(1b) blocked ⟵ true;
(1c) while waiting[i] and blocked do
(1d)       blocked ⟵ Test&Set(Reg);
(1e) waiting[i] ⟵ false;

(2) P_i executes the critical section (CS) after the entry section

(3) P_i executes the following exit section after the CS:
(3a) next ⟵ (i + 1) mod n;
(3b) while next ≠ i and waiting[next] = false do
(3c)       next ⟵ (next + 1) mod n;
(3d) if next = i then
(3e)       Reg ⟵ false;
(3f) else waiting[j] ⟵ false;

(4) P_i executes the remainder section after the exit section

until false;
```

Figure 17: Mutual exclusion with bounded waiting, using $Test\&Set$. Code shown is for process Pi, $1 \leq i \leq n$.

the other $n - f$ processes can complete all their operations in a finite number of steps, independent of the states of the f processes that may fail.

- When $f = n - 1$, any process is guaranteed to be able to complete its operations in a finite number of steps, independent of all other processes. A process does not depend on other processes, and its execution is therefore said to be *wait-free*. Wait-freedom provides independence from the behavior of other processes, and is therefore a very desirable property.

In the remainder of this chapter which deals with shared register accesses, only wait-free solutions are considered.

6.5 Register Hierarchy and Wait-free Simulations

Observe from our analysis of DSM consistency models that an underlying assumption was that any memory access takes a finite time interval, and the operation, whether a *Read*

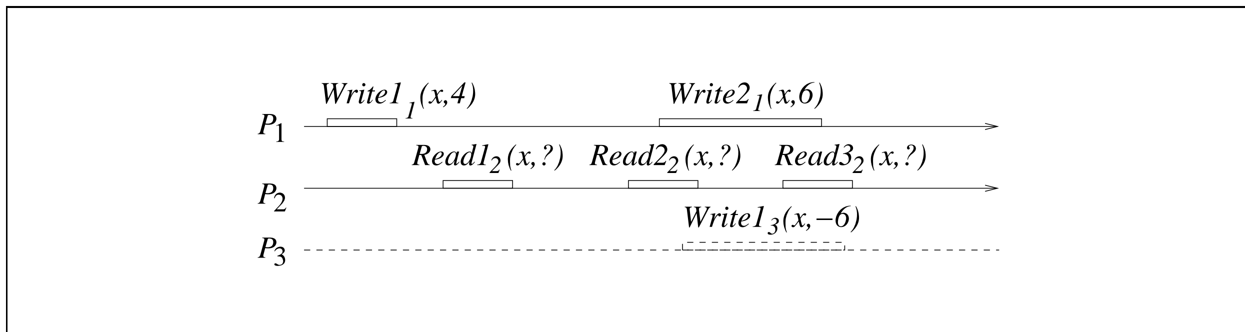

Figure 18: Examples to illustrate definitions of *safe*, *regular*, and *atomic* registers. The regular lines assume a SRSW register. If the dashed line is also used, the register is assumed to be SRMW.

or *Write*, takes effect at some point during this time duration. In the face of concurrent accesses to a memory location, we cannot predict the outcome. In particular, in the face of a concurrent *Read* and *Write* operation, the value returned by the *Read* is unpredictable. This observation is true even for a simpler multiprocessor memory, without the context of a DSM. This observation led to the research area that tried to define the properties of access orderings to the most elementary memory unit, hereafter called a *register*. The access orderings depend on the properties of the register. An implicit assumption is that of the availability of global time. This is a reasonable assumption because we are studying access to a single register. Whether that register value is replicated in the system or not is a lower detail that is not relevant to the level of abstraction of this analysis.

In keeping with the semantics of the *Read* and *Write* operations, the following register types have been identified to specify the value returned to a *Read* in the face of a concurrent *Write* operation. For the time being, we assume that there is a single reader process and a single writer process.

Safe register: A *Read* operation that does not overlap with a *Write* operation returns the most recent value written to that register. A *Read* operation that does overlaps with a *Write* operation returns *any one* of the values that the register could possibly contain at any time.

Consider the example of Figure 18 which shows several operations on an integer-valued register. We consider two cases, without and with the *Write* by P_3.

No *Write* by P_3: If the register is *safe*, $Read1_2$ must return the value 4, whereas $Read2_2$ and $Read3_2$ can return any possible integer (up to MAXINT) because these operations overlap with a *Write*, and the value returned is therefore ambiguous.

***Write* by P_3:** Same as for the "no *Write*" case.

If multiple writers are allowed, or if *Write* operations are allowed to be pipelined, then what defines the most recent value of the register in the face of concurrent *Write* operations becomes complicated. We explicitly disallow pipelining in this model and analysis. In the face of *Write* operations from different processors that overlap in time, the notion of a *serialization point* is defined. Observe that each *Write* or *Read* operation has a finite duration between its invocation and its response. In this duration, there

Type	Value	Writing	Reading
safe	binary	Single-Writer	Single-Reader
regular	integer	Multi-Writer	Multi-Reader
atomic			

Table 2: Classification of registers by Type, Value, Writing Access, and Reading Access. The strength of the register increases down each column.

is effectively a single time instant at which the operation takes effect. For a *Read* operation, this instant is the one at which the instantaneous value is selected to be returned. For a *Write* operation, this instant is the one at which the value written is first 'reflected' in the register. Using this notion of the serialization point, the 'most recent' operation is unambiguously defined.

Regular register: In addition to being a *safe* register, a *Read* that is concurrent with a *Write* operation returns either the value before the *Write* operation, or the value written by the *Write* operation.

In the example of Figure 18, we consider the two cases, with and without the *Write* by P_3.

No *Write* by P_3: $Read1_2$ must return 4, whereas $Read2_2$ can return either 4 or 6, and $Read3_2$ can also return either 4 or 6.

***Write* by P_3:** $Read1_2$ must return 4, whereas $Read2_2$ can return either 4 or -6 or 6, and $Read3_2$ can also return either 4 or -6 or 6.

Atomic register: In addition to being a *regular* register, the register is linearizable ((defined in Section 6.2.1) to a sequential register.

In the example of Figure 18, we consider the two cases, with and without the *Write* by P_3.

No *Write* by P_3: $Read1_2$ must return 4, whereas $Read2_2$ can return either 4 or 6. If $Read2_2$ returns 4, then $Read3_2$ can return either 4 or 6, but if $Read2_2$ returns 6, then $Read3_2$ must also return 6.

***Write* by P_3:** $Read1_2$ must return 4, whereas $Read2_2$ can return either 4 or -6 or 6, depending on the serialization points of the operations.

1. If $Read2_2$ returns 6 and the serialization point of $Write1_3$ precedes the serialization point of $Write2_1$, then $Read3_2$ must return 6.
2. If $Read2_2$ returns 6 and the serialization point of $Write2_1$ precedes the serialization point of $Write1_3$, then $Read3_2$ can return +6 or -6.
3. Cases (3) and (4) where $Read2_2$ returns -6 are similar to cases (1) and (2).

The following properties, summarized in Table 2, characterize registers.

- whether the register is single-valued (boolean) or multi-valued
- whether the register is a single-reader (SR) or multi-reader (MR) register

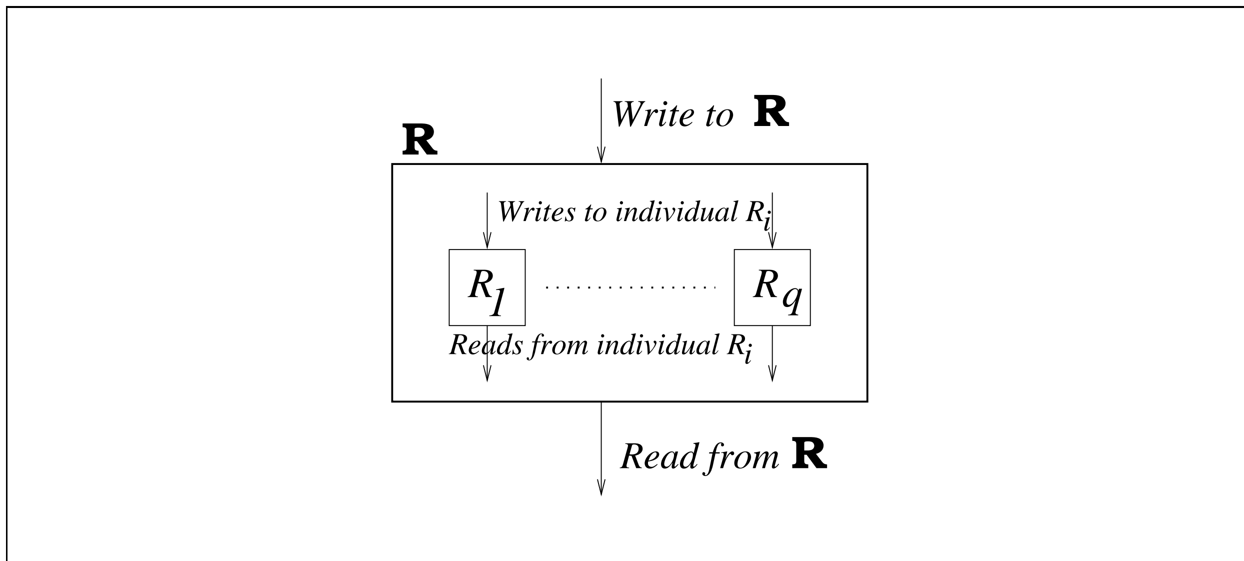

Figure 19: Register simulations.

- whether the register is a single-writer (SW) or multi-writer (MW) register
- whether the register is *safe*, *regular*, or *atomic*

The above characteristics lead to a hierarchy of 24 register types, with the most elementary being the boolean SRSW safe register and the most complex being the multi-valued MRMW atomic register.

A study of *register construction* deals with designing the more complex registers using simpler registers. Such constructions allow us to construct any register type from the most elementary register – the boolean SRSW safe register. We will study such constructions by assuming the following convention. $R_1 \ldots R_q$ are q registers that are used to construct a stronger register R, as shown in Figure 19. We assume n processes exist; note that fo various construcitons, q may be different from n.

Although the traditional memory architecture, based on serialized access via memory ports to a memory location, does not require such an elaborate classification, the bigger picture needs to be kept in mind. In addition to illustrating algorithmic design techniques, this study paves the way for accommodating newer technologies such as – quantum computing and DNA computing – for constructing system memory.

6.5.1 Construction 1: SRSW Safe to MRSW Safe

Figure 20 gives the construction of a MRSW *safe* register R using only SRSW *safe* registers. Assume the single writer is process P_0 and the n reader processes are P_1 to P_n. Each of the n processes P_i can read only SRSW register R_i. As multiple readers are not allowed to access the same register, in essence, the data needs to be replicated. So in the construction, the writer P_0 writes the same value to the n registers. Register R_i is read by P_i. In Figure 19, the value of q would hence be n. When a *Read* by P_i and a *Write* by P_0 do not overlap their access to R_i, the *Read* obtains the correct value. When a *Read* by P_i and a *Write* by P_0 overlap their access to R_i, as R_i is a *safe* register, P_i reads a legitimate value from R_i.

Complexity: This construction has a space complexity of n times the size of a single

```
(shared variables)
SRSW safe registers R_1 ... R_n <-- 0;          // R_i is readable by P_i, writable by P_0

(1)  Write(R, val) executed by single writer P_0
(1a) for all i ∈ {1 ... n} do
(1b)      R_i <-- val.

(2)  Read_i(R, val) executed by reader P_i, 1 <= i <= n
(2a) val <-- R_i
(2b) return(val).
```

Figure 20: Construction 1: SRSW Safe register to MRSW Safe register R. This construction can also be used for SRSW Regular register to MRSW Regular register R.

register, which may be either binary or integer-valued. The time complexity is n steps.

6.5.2 Construction 2: SRSW Regular to MRSW Regular

This construction is identical to Construction 1 (Figure 20) except that *regular* registers are used instead of *safe* registers. When a *Read* by P_i and a *Write* by P_0 do not overlap their access to R_i, the *Read* obtains the correct value. When a *Read* by P_i and a *Write* by P_0 overlap their access to R_i, as R_i is a *regular* register, P_i reads a legitimate value from R_i.

Complexity: This construction has a space complexity of n times the size of a single register, which may be either binary or integer-valued. The time complexity is n steps.

6.5.3 Construction 3: Boolean MRSW Safe to integer-valued MRSW Safe

Figure 21 gives the construction of an integer-valued MRSW *safe* register R. Assume the single writer is process P_0 and the n reader processes are P_1 to P_n. The construction can use only boolean MRSW registers – to construct an integer register of size m, at least $log(m)$ boolean registers are necessary. So in the construction, the writer P_0 writes the value in its binary notation to the $log(m)$ registers R_1 to $R_{log(m)}$. Similarly, any reader reads registers R_i to $R_{log(m)}$. When a *Read* by P_i and a *Write* by P_0 do not overlap, the *Read* obtains the correct value. When a *Read* by P_i and a *Write* by P_0 overlap their access to the registers, as the R_i ($i = 1$ to $log(m)$) registers are *safe*, P_i reads a legitimate value.
Complexity: This construction has a space complexity of $log(m)$ times the size of an integer m. The time complexity is $O(log(m))$ steps.

6.5.4 Construction 4: Boolean MRSW Safe to boolean MRSW Regular

Figure 22 gives the construction of a boolean MRSW *regular* register R from a MRSW *safe* register. Assume the single writer is process P_0 and the reader process is P_i ($1 \leq i \leq n$). In Figure 19, q has the value 1. P_0 writes R_1 and all the n processes read R_1.

When a *Read* by P_i and a *Write* by P_0 do not overlap, the *Read* obtains the correct value. When a *Read* by P_i and a *Write* by P_0 overlap, the *safe* register may not necessarily return

```
(shared variables)
boolean MRSW safe registers R_1 ... R_log(m) <-- 0;      // R_i readable by all, writable by P_0.

(local variable)
array of boolean: Val[1 ... log(m)];

(1) Write(R, Val[1 ... log m]) executed by single writer P_0
(1a) for all i ∈ {1 ... log(m)} do
(1b)       R_i <-- Val[i].

(2) Read_i(R, Val[1 ... log(m)]) executed by reader P_i, 1 <= i <= n
(2a) for all j ∈ {1 ... log m} do Val[j] <-- R_j
(2b) return(Val[1 ... log(m)]).
```

Figure 21: Construction 3: boolean MRSW Safe register to integer-valued MRSW Safe register R.

```
(shared variables)
boolean MRSW safe register: R' <-- 0;      // R' is readable by all, writable by P_0.

(local variables)
boolean local to writer P_0: previous <-- 0;

(1) Write(R, val) executed by single writer P_0
(1a) if previous ≠ val then
(1b)       R' <-- val;
(1c)       previous <-- val.

(2) Read(R, val) process P_i, 1 <= i <= n
(2a) val <-- R';
(2b) return(val).
```

Figure 22: Construction 4: boolean MRSW Safe register to boolean MRSW Regular register R.

the overlapping or the previous value (as required by a *regular* register), but may return a value written much earlier. If the value written before the *Read* begins is α, and the value being written by the concurrent *Write* is also α, the *Read* could return α or $(1-\alpha)$ from the *safe* register, which is a problem for the *regular* register. The solution bypasses this problem by having the *Write* use a local variable *previous* to track the previous value of *val*. If the previous value that was written (line (1b)) and stored in *previous* (line (1c)) is the same as the new value to be written, then the new value is simply not written. This avoids any concurrent access to R.

Complexity: This construction uses $O(1)$ space and time.

Can the above construction also construct a binary SRSW *atomic* register from a *safe* register? No. Consider P_1 issues a $Write1_1(\alpha)$ that completes; then $Write2_1(1-\alpha)$ begins

```
(shared variables)
boolean MRSW regular registers R_1 ... R_{m-1} ⟵ 0; R_m ⟵ 1;
                                         // R_i readable by all, writable by P_0.
(local variables)
integer: count;

(1) Write(R, val) executed by writer P_0
(1a) R_val ⟵ 1;
(1b) for count = val − 1 down to 1 do
(1c)       R_count ⟵ 0.

(2) Read_i(R, val) executed by P_i, 1 ≤ i ≤ n
(2a) count = 1;
(2b) while R_count = 0 do
(2c)       count ⟵ count + 1;
(2d) val ⟵ count;
(2e) return(val).
```

Figure 23: Construction 5: boolean MRSW Regular register to integer-valued MRSW Regular register R.

and overlaps with $Read1_2$ and $Read2_2$ of P_2. With the above construction, $Read1_2$ could return $1 - \alpha$ whereas the later $Read2_2$ could return α, thus violating the property of an *atomic* register.

6.5.5 Construction 5: Boolean MRSW Regular to integer-valued MRSW Regular

Figure 23 gives the construction of an integer-valued MRSW *regular* register R using boolean MRSW *regular* registers. Assume the single writer is process P_0 and the n reader processes are P_1 to P_n. The construction can use only boolean MRSW registers – to construct an integer register of size m, unary notation is used, so m boolean registers are necessary. In Figure 19, $q = m$, and all the n processes all the q registers.

When a *Read* by P_i and a *Write* by P_0 do not overlap, the *Read* obtains the correct value. To deal with a *Read* by P_i and a *Write*(s) by P_0 overlapping their access to the registers, the following approach is used. A reader P_i scans left-to-right looking for a '1' whereas the P_0 writer process writes '1' to the R_{val} location and then zeros out entries right-to-left. The *Read* is guaranteed to see a '1' written by one of the *Write* operations it overlaps with, or the '1' written by the *Write* that completed just before the *Read* began. As each of the bits are regular, its current or previous value is read; if the value is '0', it is guaranteed that a '1' has been written to the right. An implicit assumption here is the integer size, bounded by the number of bits in use. The register is initialized by this largest value.

Complexity: This construction uses m binary registers, where m is the largest integer that can be written by the application.

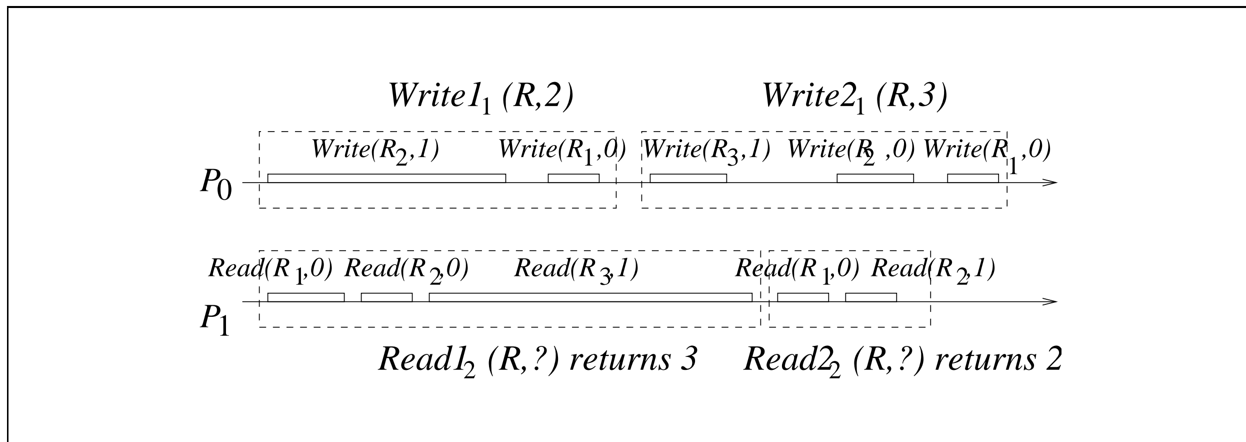

Figure 24: Example to illustrate inversion of values read.

6.5.6 Construction 6: Boolean MRSW Regular to integer-valued MRSW Atomic

Can the above construction (Figure 23) also construct a binary SRSW *atomic* register from a *regular* register? No. The problem is that when two successive *Read* operations overlap *Write* operations, 'inversion' of values returned by the *Read* operations can occur.

Consider the following sequence of operations, depicted in Figure 24.

1. $Write1_1(R, 2)$: The low-level operation $Write(R_2, 1)$ begins, i.e., $R_2 \longleftarrow 1$ begins.

2. $Read1_2(R, ?)$: The following low-level operations get executed. $count \longleftarrow 1$; $Read(R_{count}, 0)$; $count \longleftarrow 2$; $\text{Read}_2(R_{count}, 0)$; $count \longleftarrow 3$;

3. $Write1_1(R, 2)$: The low-level operation $Write(R_2, 1)$ from step 1 completes, i.e., the value '1' gets written to R_2; then the left scan to zero out R_1 proceeds by executing $Write_1(R_1, 0)$ and the $Write1_1(R, 2)$ ends.

4. $Write2_1(R, 3)$: The low-level operation $Write_1(R_3, 1)$ executes, i.e., $R_3 \longleftarrow 1$ begins and ends.

5. $Read1_2(R, ?)$: The low-level operation $Read_2(R_{count=3}, ?)$ that was to begin after step 2 returns 1; the high-level *Read* completes.

6. $Read2_2(R, ?)$: This operation's left-to right scan for a '1' finds $R_2 = 1$ and returns 2; because the low-level operation $Write2_1(R_2, 0)$ belonging to the high-level operation $Write2_1(R, 3)$ has not yet zeroed out R_2.

Here, $Read2_2(R, 2)$ returns the value written by $Write1_1(R, 2)$; whereas the earlier $Read1_2(R, 3)$ returns the value written by the later $Write2_1(R, 3)$. Hence, this execution is not linearizable.

Figure 25 gives the construction of a integer-valued MRSW *atomic* register R by modifying the above solution as follows. The reader makes a right-to-left scan for a '1' after its left-to-right scan completes. If it finds a '1' in a lower index, it updates the value to be returned to this index. The purpose is to make sure that the lowest index (say α) in which a '1' is found in this second 'right-to-left' scan is returned by the *Read*. As the writer also zeros out entries 'right-to-left', it is not possible that a later *Read* will find a '1' written earlier in a position lower than α, by a *Write* that occurred earlier than the *Write* which wrote α.

This allows a linearizable execution. In Figure 19, $q = m$, and all the n processes all the q registers.

A formal argument that this construction is correct needs to show that any execution is linearizable. To do so, it would define the *linearization point* of a *Read* and *Write* operation to capture the notion of the exact instant at which that operation effectively appears to take effect.

- The *value of the MRSW register* at any moment is x, where $R_x = 1$ and $\forall y < x$, $R_y = 0$.
- The linearization point of a *Write*(R, x) operation is the first instant (line (1a) or (1c)) when $R_x = 1$ and $\forall y < x$, $R_y = 0$.
- The linearization point of a *Read*(R, val) that returns (x) is the first instant (line (2d) or (2g)) when val gets assigned x in the low-level operations.

The following observation can now be made from the construction and the definition of the linearization point of a *Write*.

- The *value of the MRSW register* remains unchanged between the linearization points of any two consecutive *Write* operations.

The *Write* operations are naturally ordered in the linearization sequence. In order to determine a complete linearization of the *Read* operations in addition to the *Write* operations, observe the following.

- A *Read* operation returns the value written by that *Write* operation which has the latest linearization point that precedes the *Read* operation's linearization point.

It naturally follows that a later *Read* will never return the value written by a earlier *Write*, and hence the construction is linearizable.

Complexity: This construction uses m binary registers, where m is the largest integer that is written by the application program. The time complexity is $O(m)$.

6.5.7 Construction 7: Integer MRSW Atomic to integer MRMW Atomic

We are given MRSW *atomic* registers, i.e., each register has only a single writer. To simulate a MRMW *atomic* register R, the variable has multiple copies, $R_1 \ldots R_n$, one per writer process. Writer P_i can only write to its copy R_i. Reader P_i can read all the registers $R_1 \ldots R_n$. When concurrent updates occur, a global linearization order must be created somehow. The *Read* operations must be able to recognize such a global order, and then return the appropriate version as per the semantics of the atomic register. That is the challenge.

The construction is shown in Figure 26. In Figure 19, $q = n$, and all the n processes all the q MRSW registers but only P_i can write to R_i. The idea used is similar to that used by the Bakery algorithm for mutual exclusion (Section 6.3.1), wherein each process competing for the critical section first sets its flag (behaving as the writer process) signalling its intention. The competing processes that make concurrent accesses (behaving as the reader processes) then read all the flags and deduce a global order that resolves the contention.

```
(shared variables)
boolean MRSW regular registers R_1 ... R_{m-1} ⟵ 0; R_m ⟵ 1.
                                        // R_i readable by all; writable by P_0.

(local variables)
integer: count, temp;

(1) Write(R, val) executed by P_0
(1a) R_val ⟵ 1;
(1b) for count = val − 1 down to 1 do
(1c)        R_count ⟵ 0.

(2) Read_i(R, val) executed by P_i, 1 ≤ i ≤ n
(2a) count ⟵ 1;
(2b) while R_count = 0 do
(2c)        count ⟵ count + 1;
(2d) val ⟵ count;
(2e) for temp = count down to 1 do
(2f)        if R_temp = 1 then
(2g)              val ⟵ temp;
(2h) return(val).
```

Figure 25: Construction 6: boolean MRSW Atomic register to integer-valued MRSW Atomic register R.

Each register R_i has two fields: $R_i.data$ and $R_i.tag$, where $tag = \langle seq_no, pid \rangle$. A *lexicographic order* is defined on the tags, using *seq_no* as the primary key, and then *pid* as the secondary key. A common procedure invoked by the readers and writers is the *Collect* which reads all the registers, in no particular order. The reader returns the data corresponding to the (lexicographically) most recent *Write*. A writer chooses a tag greater than the (lexicographically) greatest tag returned by the *Collect*, when it writes its new value.

All the *Write* operations are lexicographically totally ordered. Each *Read* is ordered so that it immediately follows that *Write* with the matching tag. Thus, this execution is linearizable.

Complexity: This construction uses m binary registers, where m is the largest integer written by the application. The time complexity is $O(m)$.

6.5.8 Construction 8: Integer SRSW Atomic to integer MRSW Atomic

We are given SRSW *atomic* registers. To simulate a MRSW *atomic* register R, the variable has multiple copies, $R_1, \ldots R_n$, one per reader process. The single writer can write to all of these registers.

A first attempt at this construction would have the writer write to all the registers $R_1 \ldots R_n$, whereas reader P_i reads R_i. In Figure 19, $q = n$, and each R_i is read by P_i and written to by the single writer P_0. However, such a construction does not give a linearizable execution. Consider two reads *Read1*$_i$ and *Read2*$_j$ that both overlap a *Write* and *Read2*

(shared variables)
MRSW atomic registers of type $\langle data, tag \rangle$, where $tag = \langle seq_no, pid \rangle$: $R_1 \ldots R_n$;

(local variables)
array of MRSW atomic registers of type $\langle data, tag \rangle$, where $tag = \langle seq_no, pid \rangle$: $Reg_Array[1 \ldots n]$;
integer: seq_no, j, k;

(1) $Write_i(R, val)$ executed by P_i, $1 \leq i \leq n$
(1a) $Reg_Array \longleftarrow Collect(R_1, \ldots, R_n)$;
(1b) $seq_no \longleftarrow \max(Reg_Array[1].tag.seq_no, \ldots Reg_Array[n].tag.seq_no) + 1$;
(1c) $R_i \longleftarrow (val, \langle seq_no, i \rangle)$.

(2) $Read_i(R, val)$ executed by P_i, $1 \leq i \leq n$
(2a) $Reg_Array \longleftarrow Collect(R_1, \ldots, R_n)$;
(2b) **identify** j such that for all $k \neq j$, $Reg_Array[j].tag > Reg_Array[k].tag$;
(2c) $val \longleftarrow Reg_Array[j].data$;
(2d) **return**(val).

(3) $Collect(R_1, \ldots, R_n)$ invoked by *Read* and *Write* routines
(3a) **for** $j = 1$ **to** n **do**
(3b) $Reg_Array[j] \longleftarrow R_j$;
(3c) **return**(Reg_Array).

Figure 26: Construction 7: integer MRSW Atomic register to integer MRMW Atomic register R.

begins after *Read1* terminates. It is possible that:

1. $Read1_i$ reads R_i after the *Write* has written to R_i
2. but $Read2_j$ reads R_j before the writer has had a chance to update R_j.

This results in a non-linearizable execution.

The problem above arose because a reader did not have access to what other readers read; in particular, a reader P_i cannot tell if another *Read* by P_j that completed before this *Read* began got a value that is newer than the value that the writer has written to R_i. In fact, performing multiple reads by the P_i processes, and/or more writes by P_0, and/or using more registers cannot solve this problem.

To fix this problem, a reader process P_i must choose the latest of the values that other reader processes have last read, and the value in R_i. As only SRSW registers are available, unfortunately, this requires communication between each pair of reader processes, leading to $O(n^2)$ variables. Thus, a reader process must also write! An array $Last_Read_Values[1 \ldots n, 1 \ldots n]$ is used for this purpose. $Last_Read_Values[i, j]$ is the value that P_i's last *Read* returned, which P_i has set aside for P_j to know about. Once a reader P_i determines the latest of the values that other readers read (lines 2(b-d)), and the value written for it by the writer process (line 2a), the reader publishes this value in $Last_Read_Values[i, *]$ (lines 2e-2f). As there is

(shared variables)
SRSW atomic register of type $\langle data, seq_no \rangle$, **where** $data, seq_no$ **are integers**: $R_1 \ldots R_n \longleftarrow \overline{\langle 0, 0 \rangle}$;
SRSW atomic register array of type $\langle data, seq_no \rangle$, **where** $data, seq_no$ **are integers**: $Last_Read_Values[1 \ldots n, 1 \ldots n] \longleftarrow \overline{\langle 0, 0 \rangle}$;

(local variables)
array of $\langle data, seq_no \rangle$: $Last_Read[0 \ldots n]$;
integer: $seq, count$;

(1) $Write(R, val)$ executed by writer P_0
(1a) $seq \longleftarrow seq + 1$;
(1b) **for** $count = 1$ **to** n **do**
(1c) $R_{count} \longleftarrow \langle val, seq \rangle$. // write to each SRSW register

(2) $Read_i(R, val)$ executed by P_i, $1 \leq i \leq n$
(2a) $\langle Last_Read[0].data, Last_Read[0].seq_no \rangle \longleftarrow R_i$; // $Last_Read[0]$ stores value of R_i
(2b) **for** $count = 1$ **to** n **do** // read into $Last_Read[count]$, the latest values stored for P_i by P_{count}
(2c) $\langle Last_Read[count].data, Last_Read[count].seq_no \rangle \longleftarrow$
$\langle Last_Read_Values[count, i].data, Last_Read_Values[count, i].seq_no \rangle$;
(2d) **identify** j such that for all $k \neq j$, $Last_Read[j].seq_no \geq Last_Read[k].seq_no$;
(2e) **for** $count = 1$ **to** n **do**
(2f) $\langle Last_Read_Values[i, count].data, Last_Read_Values[i, count].seq_no \rangle \longleftarrow$
$\langle Last_Read[j].data, Last_Read[j].seq_no \rangle$;
(2g) $val \longleftarrow Last_Read[j].data$;
(2h) **return**(val).

Figure 27: Construction 8: integer SRSW Atomic register to integer MRSW Atomic register R.

a single writer, the format $\langle data, seq_no \rangle$ for each register value and each $Last_Read_Value$ entry is adequate to give a total order on all the values written by it. The construction is shown in Figure 27. Here, $q = n^2 + n$ – there are n^2 SRSW registers that act as personalized mailboxes between pairs of processes, and the n registers that are the mailboxs between writer P_0 and each reader P_i.

Complexity: This construction uses $O(n^2)$ integer registers. The time complexity is $O(n)$.

Achieving linearizability: All the *Write* operations form a total order. A *Read* by P_i returns the value of the latest preceding *Write*, as observed directly from the register R_i, or indirectly from the register R_j and communicated to P_i via $Last_Read_Values$. In a linearized execution, a *Read* is placed after the *Write* whose value it reads. For nonoverlapping *Reads*, their relative order represents the order in a linearizable execution, because of the indirect communication among readers. For overlapping *Read*s, their ordering in a linearized execution is consistent with the *Write*s whose values they read. Hence, the construction is a valid construction.

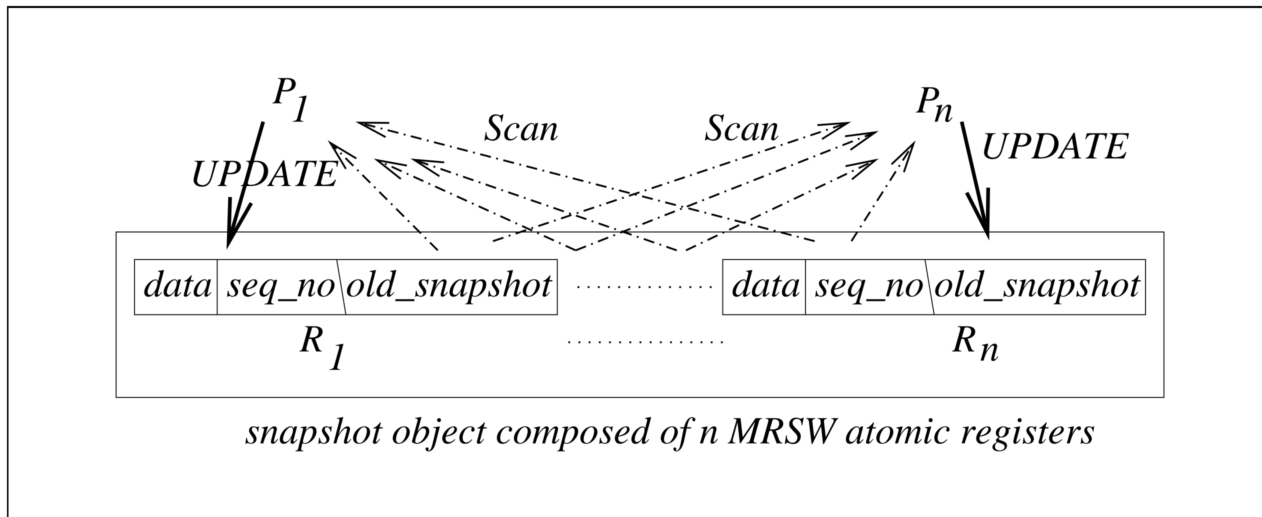

Figure 28: Atomic snapshot object, using MRSW atomic registers.

6.6 Wait-free Atomic Snapshots of Shared Objects

Observing the global state of a distributed system is a fundamental problem. For message-passing systems, we have studied how to record global snapshots which represent an instantaneous possible global state that could have occurred in the execution. The snapshot algorithms used message-passing of control messages, and were inherently inhibition-free, although some variants that use fewer control messages do require inhibition.

In this section, we examine the counterpart of the global snapshot problem in a shared-memory system, where only *Read* and *Write* primitives can be used. The problem can be modeled as follows.

Given a set of SWMR atomic registers $R_1 \ldots R_n$, where R_i can be written only by P_i and can be read by all processes, and which together form a compound high-level object, devise a *wait-free* algorithm to observe the state of the object at some instant in time. The following actions are allowed on this high-level object, as also illustrated in Figure 28.

- $Scan_i$: This action invoked by P_i returns the atomic snapshot which is an instantaneous view of the object $(R_1, \ldots, R_n)$ at some instant between the invocation and termination of the *Scan*.
- $Update_i(val)$: This action invoked by P_i writes the data val to register R_i.

Clearly, any kind of locking mechanism is unacceptable because it is not wait-free. Consider the following attempt at a wait-free solution. The format of each register R_i is assumed to be the tuple: $\langle data, seq_no \rangle$ in order to uniquely identify each *Write* operation to the register. A scanner would repeatedly scan the high-level object until two consecutive scans, called *double-collect* in the shared memory context, returned identical content. This principle of "double-collect" has been encountered in multiple contexts, such in two-phase deadlock detection and two-phase termination detection algorithms, and essentially embodies the two-phase observation rule. However, this solution in not wait-free because between the two observations of each double-collect, an *Update* by another process can prevent the *Scan* from being successful.

A wait-free solution is given in Figure 30. Process P_i can write to its MRSW register R_i and can read all registers $R_1, \ldots R_n$. To design a wait-free solution, it needs to be ensured that a scanner is not indefinitely prevented from getting identical scans in the double-collect, by some writer process periodically making updates. The problem arises because of the imbalance in the roles of the scanner and updater – the updater is inherently more powerful in that it can prevent all scanners from being successful. One elegant solution therefore neutralizes the unfair advantage of the updaters by forcing the updaters to follow the same rules as the scanner. Namely, the updaters also have to perform a double-collect, and only after performing a double-collect can an updater write the value it needs to! Additionally, an updater also writes the snapshot it collected in the register, along with the new value of the data item. Now, if a scanner detects that an updater has made an update after the scanner initiated its *Scan*, then the scanner can simply 'borrow' the snapshot recorded by the updater in its register. The updater helps the scanner to obtain a consistent value. This is the principle of "helping" that is often used in designing wait-free solutions for various problems.

A scanner detects that an updater has made an update after the scanner initiated its *Scan*, by using the local array *changed*. This array is reset to 0 when the *Scan* is invoked. Location *changed*[k] is incremented (line (2k)) if the *Scan* procedure detects (line (2j)) that process P_k has changed its *data* and *seq_no* (and implicitly the *old_snapshot*) fields in R_k. Based on the value of *changed*[k], different inferences can be made, as now explained with the help of Figure 29.

- If *changed*[k] $= 2$ (line (2l)), then two updates (line (1b)) were made by P_k after P_i began its *Scan*. Between the first and the second update, the *Scan* preceding the second update must have completed successfully, and the scanned value was recorded in the *old_snapshot* field. This old snapshot can be safely borrowed by the scanner P_i (line (2m)) because it was recorded after P_k finished its first double-collect, and hence after the scanner P_i initiated its *Scan*.

- However, if *changed*[k] $= 1$, it cannot be inferred that the *old_snapshot* recorded by P_k was taken after P_i's *Scan* began. When P_k does its *Update* (the first 'write' shown in Figure 29(b)), the value it writes in *old_snapshot* is only the result of a double-scan that preceded the 'write' and may be a value that existed before P_i's *Scan* began.

There are two cases by which a snapshot can be captured, as illustrated using Figure 29.

1. A scanner can collect a snapshot (line (2g)) if the double-collect (lines (2d-2e)) returns identical views (line (2f)). See Figure 29(a). The returned snapshot represents an instantaneous state that existed at all times between the end of the first *collect* (line (2d)) and the start of the second *collect* (line (2e)).

2. Otherwise the scanner returns a borrowed snapshot (line (2m)) from P_k if P_k has been noticed to have made two updates (lines (2l)) and therefore P_k has made a *Scan* embedded inside P_i's *Scan*. This borrowed snapshot itself (i) may have been obtained directly via a *double-collect*, or (ii) indirectly been borrowed from another process (line (2l)). In case (i), it represents an instantaneous state in the duration of the *double-collect*. In case (ii), a recursive argument can be applied. Observe that there are n processes, so the recursive argument can hold at most n times. The $n+1^{th}$ time, a

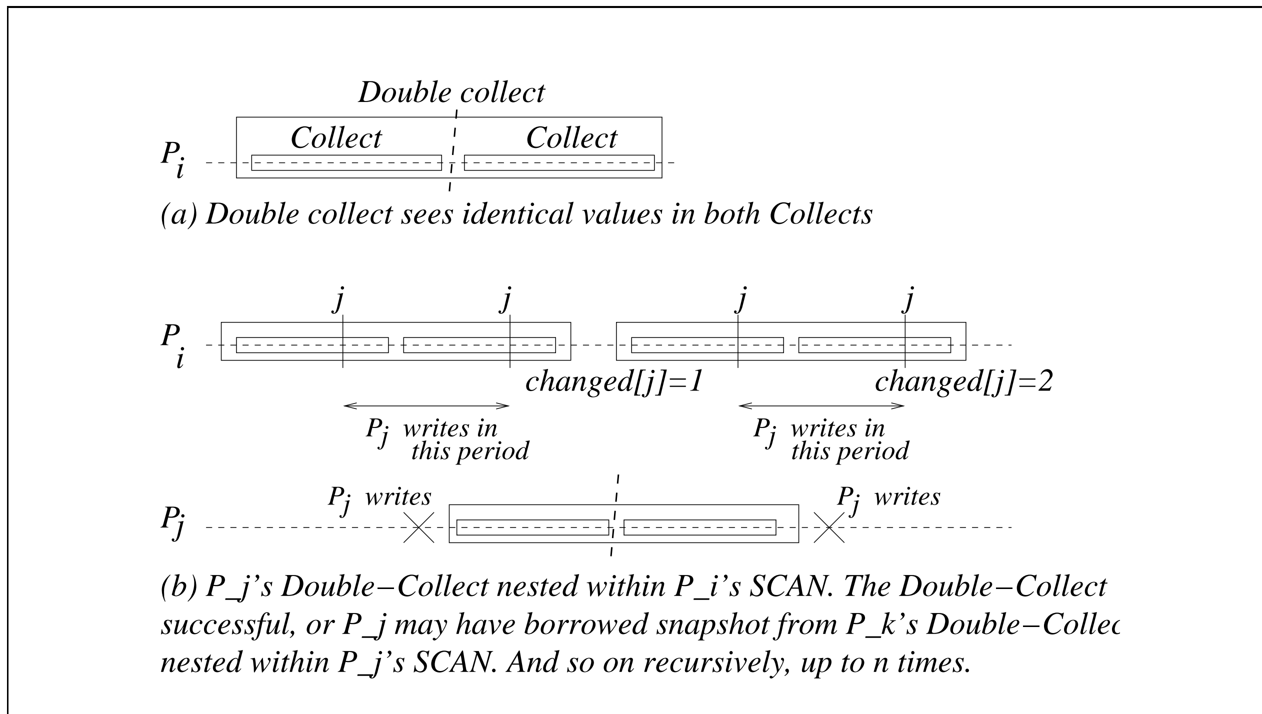

Figure 29: Nesting of double-collects, in Scanning for Atomic snapshots of object.

double-collect must have been successful. See Figure 29(b). Note that between the two *double-collect*s of P_i that are shown, there may be up to $n-2$ other unsuccessful *double-collect*s of P_i. Each of these $(n-2)$ other *double-collect*s corresponds to some P_k $k \neq i, j$, having 'changed' once.

The linearization of the *Scan* and *Update* operations follows in a straightforward manner. For example, nonoverlapping operations get linearized in the order of their occurrence. An operation by P_i that borrows a snapshot from P_k gets linearized after P_k.

Complexity: The space complexity is $O(n)$ integers. The shared space is $O(n^2)$ corresponding to each of the n registers of size $O(n)$ each. The time complexity is $O(n^2)$. This is because the main Scan loop has a complexity of $O(n)$ and the loop may be executed at most $(n+1)$ times – the $n+1$-th time, at least one process P_k must have caused P_i's local *changed*$[k]$ to reach a value of two, triggering an end to the loop (lines (2k-2l)).

6.7 Exercises

1. Why do the algorithms for sequential consistency (Section 6.2.2) not require the *Read* operations to be broadcast?

2. Give a formal proof to justify the correctness of the algorithm in Figure 7 that implements sequential consistency using local *Read* operations.

3. In the algorithm to implement sequential consistency using local *Write* operations, as given in Figure 8, why is a single counter *counter* sufficient for the algorithm's correctness?

```
(shared variables)
MRSW atomic register of type ⟨data, seq_no, old_snapshot⟩, where data, seq_no are of type
integer, and old_snapshot[1...n] is array of integer: R_1 ... R_n;

(local variables)
array of int: changed[1...n];
array of type ⟨data, seq_no, old_snapshot⟩: v1[1...n], v2[1...n], v[1...n];

(1) Update_i(x)
(1a) v[1...n] ⟵ Scan_i;
(1b) R_i ⟵ (x, R_i.seq_no + 1, v[1...n]).

(2) Scan_i
(2a) for count = 1 to n do
(2b)        changed[count] ⟵ 0;
(2c) while true do
(2d)        v1[1...n] ⟵ collect();
(2e)        v2[1...n] ⟵ collect();
(2f)        if (∀k, 1 ≤ k ≤ n)(v1[k].seq_no = v2[k].seq_no) then
(2g)              return(v2[1].data, ..., v2[n].data);
(2h)        else
(2i)              for k = 1 to n do
(2j)                    if v1[k].seq_no ≠ v2[k].seq_no then
(2k)                          changed[k] ⟵ changed[k] + 1;
(2l)                          if changed[k] = 2 then
(2m)                                return(v2[k].old_snapshot).
```

Figure 30: Wait-free atomic snapshot of a shared MRSW object.

In other words, why is a separate counter $counter_x$ not required to track the number of updates issued to each variable x, where a *Read* operation on x gets delayed only if $counter_x > 0$? If such a separate counter were used for every variable, what consistency model would be implemented?

4.
   - In Figure 9(a), analyze whether the execution is linearizable.
   - In Figure 9(b), what forms of memory consistency are satisfied if the two *Read* operations of P_4 return 7 first and then 4?

5. Give a detailed implementation of causal consistency, and provide a correctness argument for your implementation.

6. Give a detailed implementation of PRAM consistency, and provide a correctness argument for your implementation.

7. Give a detailed implementation of slow memory, and provide a correctness argument for your implementation. Is the implementation less expensive than that of PRAM consistency which is a stricter consistency model?

```
(shared variables)
boolean: turn ⟵ false;                                    // shared register initialized
array of boolean: want[0, 1];

repeat
(1) P_i executes the following for the entry section:
(1a) wanting[i] ⟵ true;
(1b) turn ⟵ 1 − i;
(1c) while wanting[1 − i] and turn = 1 − i do
(1d)      no-op;

(2) P_i executes the critical section (CS) after the entry section

(3) P_i executes the following exit section after the CS:
(3a) wanting[i] ⟵ false;

(4) P_i executes the remainder section after the exit section

until false;
```

Figure 31: Peterson's mutual exclusion for two processes $P_i = 0, 1$. Modulo=2 artithmatic is used.

8. Show that Constructions 1 and 2 (Figure 20) work for binary registers as well as integer-valued registers.

9. Why are two passes needed by the reader in Construction 6, Figure 25, for a MRSW atomic register? Why does a single right-to-left pass not suffice?

10. Peterson's mutual exclusion algortihm for two processes is shown in Figure 31.

    (a) Show that it satisfies mutual exclusion, progress, and bounded waiting.

    (b) Use this algorithm as a building block to construct a hierarchical mutual exclusion algorithm for an arbitrary number of processes. (Hint: use a logarithmic number of steps in the hierarchy.)

11. Determine the average case time complexity of the wait-free atomic snapshot of a shared object, given in Figure 30.

6.8 Bibliographic Notes

A good survey on distributed shared memory systems is given by Protic, Tomasevic, and Milutinovic [25]. This includes coverage of the various DSM systems such as Firefly, Sequent, Alewife, Dash, Butterfly, CM*, Ivy, Mirage, Midway, Munin, Linda and Orca.

The sequential consistency model was defined by Lamport [16]. The linearizability model was formalized by Lamport [18] and developed by Herlihy and Wing [10]. The implementations of linearizability and sequential consistency based on the broadcast primitive and

assuming full replication are from Attiya and Welch [5], whereas a similar implementation of sequential consistency is given by Bal, Kaashoek, and Tanenbaum [6]. The causal consistency model was proposed by [3]. The PRAM model was proposed by Lipton and Sandberg [22]. The slow memory model was proposed by Hutto and Ahamad [11]. Other consistency models such as weak consistency [8], release consistency [9], and entry consistency [7] that apply to selected instructions in the code, were developed mainly in the computer architecture research community, and are discussed in [2, 1].

The bakery algorithm for mutual exclusion was presented by Lamport [14]. The fast mutual exclusion algorithm was presented by Lamport [20]. The two-process mutual exclusion algorithm was presented by Peterson [23]. Its modification that is asked as Exercise 10 is based on the algorithm by Peterson and Fischer [24].

The notion of wait-freedom was proposed by Lamport [] and developed by Herlihy [12]. The definition and classification of registers as safe, regular, and atomic were given by Lamport [17, 18, 19]. Constructions 1 to 5 were proposed by Lamport [18]. Register Construction 6 was proposed by Vidyasankar [26]. Register Construction 7 was proposed by Vitanyi and Awerbuch [27]. Register Construction 8 was proposed by Israeli and Li [13]. A construction of a MRMR snapshot object using MRSW snapshot objects and MRMW registers was proposed by Anderson [4].

References

[1] S. Adve, K. Gharachorloo, Shared memory consistency models: A tutorial, IEEE Computer Magazine 29(12): 66-76, 1996.

[2] S. Adve, M. Hill, A unified formalization of four shared-memory models, IEEE Transactions on Parallel and Distributed Systems 4(6): 613-624, 1993.

[3] M. Ahamad, G. Neiger, J. Burns, P. Kohli, P. Hutto, Causal memory: Definitions, implementation, and programming, Distributed Computing, 9(1): 37-49, 1995.

[4] J. Anderson, Multi-writer composite registers, Distributed Computing, 7(4): 175-196, 1994.

[5] H. Attiya, J. Welch, Sequential consistency versus linearizability, ACM Transactions on Computer Systems, 12(2): 91-122, 1994.

[6] H. Bal, F. Kaashoek, A. Tanenbaum, Orca: A language for parallel programming of distributed systems, IEEE Transactions on Software Engineering, 18(3): 180-205, 1992.

[7] B. Bershad, M. Zekauskas, W. Sawdon, The Midway distributed shared memory system, CMU Technical Report CMU-CS-93-119. (Also in Proceedings of COMPCON 1993.)

[8] M. Dubois, C. Scheurich, Memory access dependencies in shared-memory multiprocessors, IEEE Transactions on Software Engineering, 16(6): 660-673, 1990.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, Proceedings of the Seventeenth International Symposium on Computer Architecture, pages 15–26, Seattle, WA, May 1990.

[10] M. Herlihy, J. Wing, Linearizability: A correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems, 12(3): 463-492, 1990.

[11] P. Hutto, M. Ahamad, Slow memory: Weakening consistency to enchance oncurrency in distributed shared memories, Proc. IEEE International Conference on Distributed Computing Systems, 302-311, 1990.

[12] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems, 13(1): 124-149, 1991.

[13] A. Israeli, M. Li, Bounded timestamps, Distributed Computing, 6(4): 205-209, 1993.

[14] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Communications of the ACM, 17(8): 453-455, 1974.

[15] L. Lamport, Proving the correctness of multiprocess programs, IEEE Transactions on Software Engineering, 3(2): 125-143, 1977.

[16] L. Lamport, How to make a multiprocessor that correctly executes multiprocess programs, IEEE Transactions on Computers, 28(9): 690-691, 1979.

[17] L. Lamport, On interprocess communication, Part I: Basic formalism, Distributed Computing, 1(2): 77-85, 1986.

[18] L. Lamport, On interprocess communication, Part II: Algorithms, Distributed Computing, 1(2): 86-101, 1986.

[19] L. Lamport, The mutual exclusion problem, Part II: Statement and solutions, Journal of the ACM, 33(2): 327-348, 1986.

[20] L. Lamport, A fast mutual exclusion algorithm, ACM Transactions on Computer Systems, 5(1): 1-11, 1987.

[21] L. Lamport, Concurrent reading and writing, Communications of the ACM, 20(11): 806-811, 1977.

[22] R. Lipton, J. Sandberg, PRAM: A scalable shared memory, Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[23] G.L. Peterson, Myths about the Mutual exclision problem, Information Processing Letters, 12: 115-116, 1981.

[24] G.L. Peterson, M. Fischer, Economical solutions for the mutual exclusion problem in a distributed system, Proceedings 9th ACM Symposium on Theory of Computing, 91-97, 1977.

[25] J. Protic, M. Tomasevic, V. Milutinovic, Distributed Shared Memory: Concepts and Systems, IEEE Computer Society Press, 1997.

[26] K. Vidyasankar, Converting Lamport's regular register to atomic register, Information Processing Letters, 28: 287-290, 1988.

 © A. Kshemkalyani, 2004

[27] P. Vitanyi, B. Awerbuch, Atomic shared register access byu asynchronous hardware, Proceedings 27th IEEE Symposium on Foundations of Computer Science, pp. 233-243, 1986.