CHAPTER 6

Covering Arguments

A process is *poised* at an object if it will apply a primitive to the object when it is next allocated a step by the scheduler. If the value of the object resulting from applying this primitive does not depend on its current value, then the information stored in the object will be obliterated. In this case, we say that the primitive is *historyless* and the process *covers* the object.

A *covering argument* is useful for proving lower bounds for asynchronous shared memory systems. The goal is to construct a configuration in which a large set of objects are covered. The size of the objects does not matter: They can be arbitrarily large. The construction is usually inductive, with the number of covered objects increasing as the argument progresses. The processes covering these objects can be used to hide information that other processes may have stored there and wished to communicate. If these objects are the only objects that the other processes have modified, then the steps of these other processes can be hidden, in the sense that if they are removed from the execution, the resulting execution is indistinguishable to the remaining processes.

For example, suppose that processes communicate through r multi-writer registers. Note that write is a historyless primitive. Consider a history, β, starting from some configuration C, in which r processes, $p_{i_1}, \ldots, p_{i_r}$, each write a value to a different register, one after the other. This is called a block write. Let γ be another history starting from C by a subset of processes Q disjoint from $\{p_{i_1}, \ldots, p_{i_r}\}$. Then the executions starting from C with histories β and $\gamma\beta$ are indistinguishable to all processes not in Q. Moreover, each register has the same value in configurations $C\beta$ and $C\gamma\beta$. Thus, γ is hidden from the processes not in Q, no matter what they do in the future.

Another example of a historyless primitive is swap. It has a single input parameter v. When applied to an object, it sets the value of the object to v and returns the value the object had beforehand. A consecutive sequence of swaps applied by different processes to different objects is called a block swap. When a set of processes performs a block write to a set of objects, they get no information about the previous values of those objects. In contrast, when a set of processes performs a block swap to a set of objects, they (collectively) learn the previous values of these objects. Hence, to hide the information that was stored there, the adversary does not let these processes take any further steps.

Any historyless primitive that sets the value of an object to v (regardless of its previous value) can be simulated by swap(v). Thus, for proving lower bounds on implementations using historyless primitives, it suffices to restrict attention to swap.

A primitive is *trivial* if it can never change the value of an object. The most common example of a trivial primitive is read. We say that an object is *historyless* if it only supports historyless and trivial primitives.

Covering arguments were introduced by Burns and Lynch, in their paper *Bounds on Shared Memory for Mutual Exclusion*, Information and Computation, volume 107, 1993, pages 171–184, to prove a lower bound on the number of registers needed to solve mutual exclusion. This result will be presented in Section 6.1. In subsequent sections, we present a variety of other covering arguments: lower bounds on the number of multi-writer registers needed to implement timestamps, space and step complexity lower bounds for the implementation of a counter using swap, a lower bound on the number of multi-writer registers needed to implement a multi-writer snapshot object, and a step complexity lower bound for their space optimal implementations. In Section 6.5, we prove a lower bound on the worst case number of stalls incurred by READ in any implementation of a counter using read-modify-write primitives. In this case, we use a covering argument to increase contention, rather than to hide information.

6.1 A Space Lower Bound for Mutual Exclusion

Our first covering argument shows that any algorithm for mutual exclusion, with $n \geq 2$ processes, uses at least n registers. Recall that the mutual exclusion problem was defined in Section 2.2. We consider an asynchronous shared memory model that contains only multi-writer registers. While a process is in the remainder section, we do not consider it to be covering any of the registers.

First, we look at a solo execution by a process that takes it from the remainder section to the critical section. We show that along the way, there is a configuration in which one more register is covered.

Lemma 6.1. *Suppose that C is a configuration in which process p_i is in its remainder section. Let α be a finite history by process p_i starting from configuration C such that p_i is in the critical section in configuration $C\alpha$. Let R be the set of registers which are covered in configuration C. Then, during α, process p_i writes to some register that is not in R.*

Proof. By contradiction. Suppose that during α, process p_i only writes to registers in R. Let β be a block write to R starting from configuration C. Then the value of every register is the same in configurations $C\beta$ and $C\alpha\beta$ and $C\beta \overset{q}{\sim} C\alpha\beta$, for all $q \neq p_i$. Note that p_i is still in the remainder section in configuration $C\beta$.

Starting from $C\beta$, we show that there exists a finite history γ by processes other than p_i such that one of these processes is in the critical section in configuration $C\beta\gamma$. If some process is in the critical section in $C\beta$, then it suffices to let γ be empty. If not, but there is some process in the trying section, then deadlock freedom implies the existence of γ. If there is no process in the trying or critical sections, but there is some process other than p_i in the remainder section, the adversary can first let some such

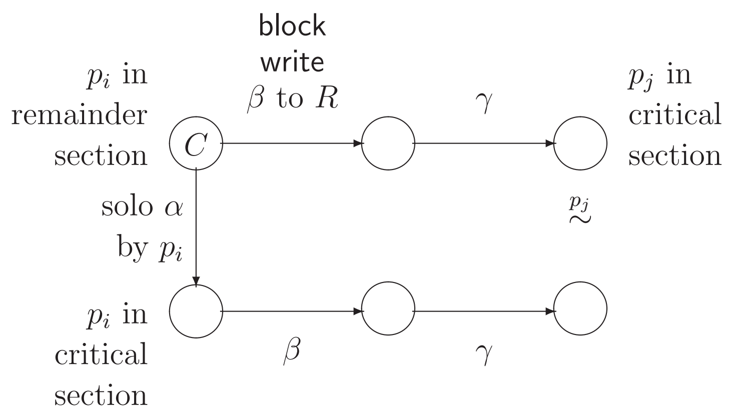

block write β to R
γ
p_i in remainder section
C
p_j in critical section
solo α by p_i
$\overset{p_j}{\sim}$
p_i in critical section
β
γ

Figure 6.1. The situation in Lemma 6.1.

process enter the trying section. Finally, if all processes other than p_i are in the exit section, the adversary can choose one of them and first allocate steps to it until it is in the remainder section. This is because a process that is in the exit section will always leave it within a bounded number of steps.

Let $p_j \neq p_i$ be a process in the critical section in configuration $C\beta\gamma$. Since $C\beta \overset{q}{\sim} C\alpha\beta$ for all $q \neq p_i$, it follows by Lemma 2.1 that $C\beta\gamma \overset{p_j}{\sim} C\alpha\beta\gamma$. This is illustrated in Figure 6.1. Note that both p_i and p_j are in their critical sections in configuration $C\alpha\beta\gamma$. This violates mutual exclusion. □

A configuration C is *reachable* from configuration Q if there is a finite history α such that $C = Q\alpha$. A configuration is *quiescent* if every process is in the remainder section, i.e. there is no process that has entered the trying section, but has not subsequently finished the exit section. From any quiescent configuration, we show how to construct a sequence of reachable configurations with successively more covered registers. Furthermore, each of these configurations will be indistinguishable from a quiescent configuration to successively fewer processes.

Lemma 6.2. *From any quiescent configuration, for $k = 1, \ldots, n$, there are reachable configurations, C and D, such that D is quiescent, each register has the same value in C and D, $p_0, \ldots, p_{k-1}$ cover k different registers in C, and $C \overset{q}{\sim} D$ for all $q \in \{p_k, \ldots, p_{n-1}\}$.*

Proof. By induction on k.

First consider $k = 1$. From any quiescent configuration Q, deadlock freedom implies that there is a solo execution by process p_0 that results in a configuration in which p_0 is in the critical section. Let α be the history of that execution.

There are no registers covered in Q, so, by Lemma 6.1, during α, process p_0 writes to some register. Let α' be the longest prefix of α that contains no writes. Then p_0 covers a register in $Q\alpha'$ and $Q\alpha' \overset{q}{\sim} Q$ for all $q \in \{p_1, \ldots, p_{n-1}\}$. Hence, the claim is true with $C = Q\alpha'$ and $D = Q$.

Now assume the claim is true for k, where $1 \leq k < n$. By the induction hypothesis, from any quiescent configuration Q_t, there are configurations C_t and D_t that satisfy the claim. Let β_t be a block write by $\{p_0, \ldots, p_{k-1}\}$ starting from C_t. Since C_t and

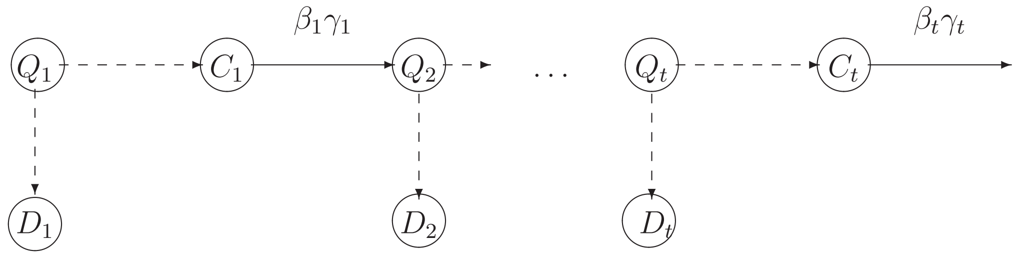


Figure 6.2. The situation in the proof of Lemma 6.2.

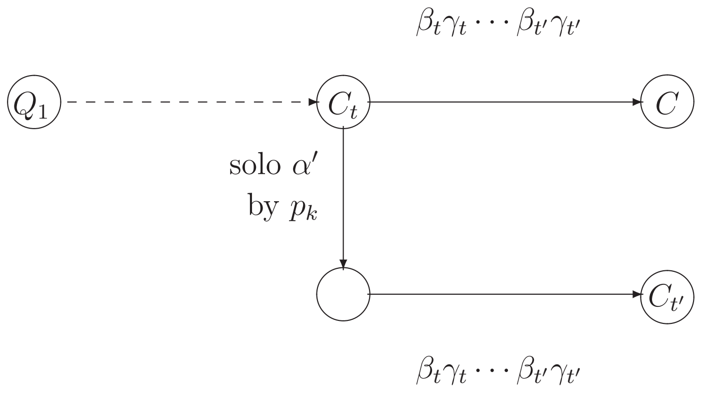
D_t are indistinguishable to the rest of the processes and D_t is quiescent, $p_k, \ldots, p_{n-1}$ are all in the remainder section in C_t. Thus, deadlock freedom implies that there is a finite history γ_t by the processes in $\{p_0, \ldots, p_{k-1}\}$ starting from $C_t\beta_t$ such that $Q_{t+1} = C_t\beta_t\gamma_t$ is quiescent.

From any quiescent configuration Q, consider the sequence of configurations $Q = Q_1, Q_2, \ldots$, as shown in Figure 6.2. By the pigeon hole principle, there exist $1 \leq t < t' \leq \binom{r}{k} + 1$ such that β_t and $\beta_{t'}$ are block writes to the same set of k registers R.

Configuration D_t is quiescent. Hence, deadlock freedom implies that there is a solo execution by process p_k, starting from D_t, that takes p_k to the critical section. Let α be the history of that execution. Since $C_t \overset{p_k}{\sim} D_t$ and each register has the same value in C_t and D_t, it follows by Lemma 2.1 that α can occur starting from C_t and p_k is in the critical section in $C_t\alpha$.

By Lemma 6.1, during α, process p_k writes to some register not in R. Let α' be the shortest prefix of α such that, in configuration $C_t\alpha'$, process p_k covers some register not in R. Since all writes in α' are to registers in R and β_t is a block write to R, each register has the same value in $C_t\beta_t$ and $C_t\alpha'\beta_t$. Since $C_t\beta_t \overset{q}{\sim} C_t\alpha'\beta_t$ for all $q \neq p_k$, Lemma 2.1 implies that $\gamma_t \cdots \beta_{t'-1}\gamma_{t'-1}$ can occur starting from $C_t\alpha'\beta_t$ and $C = C_t\alpha'\beta_t\gamma_t \cdots \beta_{t'-1}\gamma_{t'-1} \overset{q}{\sim} C_{t'}$ for all $q \neq p_k$. This is illustrated in Figure 6.3. Since $C_{t'} \overset{q}{\sim} D_{t'}$, for all $q \in \{p_k, \ldots, p_{n-1}\}$, it follows that $C \overset{q}{\sim} D_{t'}$ for all $q \in \{p_{k+1}, \ldots, p_{n-1}\}$.

Furthermore, each register has the same value in C, $C_{t'}$, and $D_{t'}$. Since p_k takes no steps during $\beta_t\gamma_t \cdots \beta_{t'-1}\gamma_{t'-1}$, it covers the same register in configuration C. Thus,



$\beta_t\gamma_t \cdots \beta_{t'}\gamma_{t'}$
Q_1
C_t
C
solo α'
by p_k
$C_{t'}$
$\beta_t\gamma_t \cdots \beta_{t'}\gamma_{t'}$

Figure 6.3. Two indistinguishable executions in the proof of Lemma 6.2.

$p_0, \ldots, p_{k-1}, p_k$ cover $k+1$ different registers in C. Therefore the claim is true for $k+1$ with $D = D_{t'}$. Hence, by induction, it is true for $k = 1, \ldots, n$. □

The lower bound follows easily from this lemma with $k = n$, using the fact that the initial configuration is quiescient.

Theorem 6.3. *Any mutual exclusion algorithm for $n \geq 2$ processes must use at least n registers.*

6.2 Space Lower Bounds for the Implementation of Timestamps

Timestamps provide information about the temporal ordering of events in an execution. Formally, a *timestamp* system consists of a partially ordered set $(U, <)$ of timestamps and an operation GetTS with range U that satisfies the following validity condition: If one GetTS operation is completed before another GetTS operation begins and these operations return t_1 and t_2, respectively, then $t_1 < t_2$. If the two operations are concurrent, then t_1 can be less than, greater than, or incomparable to t_2.

The model we consider is asynchronous shared memory in which n processes communicate via multi-writer registers. We show that n is a lower bound on the number of registers needed for implementing a certain class of timestamp systems.

Theorem 6.4. *Any n-process implementation of* GetTS *with range $\mathbb{N}$, under its usual ordering, and that satisfies solo termination requires at least n registers.*

Proof. We prove by induction that, for $i = 0, \ldots, n$, there is a reachable configuration C_i in which a set P_i of i processes cover a set R_i of i different registers. The theorem follows from this claim with $i = n$.

When $i = 0$, let C_0 be the initial configuration and let $P_0 = R_0 = \phi$. Now, let $1 \leq i \leq n$ and suppose that there is a reachable configuration C_{i-1} in which a set P_{i-1} of $i-1$ processes cover a set R_{i-1} of $i-1$ different registers. If $i > 1$, let $p \in P_{i-1}$; otherwise, let p be any process.

Consider an execution that starts from configuration C_{i-1} with a block write β by the processes in P_{i-1} to the registers of R_{i-1}, followed by a solo execution γ by p in which p completes its pending operation, if any, and then performs GetTS. By solo termination, this operation eventually completes and returns some timestamp t.

Let q be a process not in $P_{i-1} \cup \{p\}$. We now show that a solo execution by q, starting from C_{i-1}, in which it repeatedly performs GetTS, must eventually write to a register not in R_{i-1}. Note that, by solo termination, each instance of GetTS in this solo execution will eventually terminate. Let t_j be the timestamp returned by the j'th instance of GetTS by q in this solo execution. Then $t_j < t_{j+1}$ for all $j \geq 1$. There are only a finite number of timestamps less than t, so there exists j such that $t_j \not< t$.

To derive a contradiction, suppose that q does not write to any register outside R_{i-1} during the solo execution, α, of its first j instances of GetTS, starting from C_{i-1}. Since each register has the same value in configurations $C_{i-1}\beta$ and $C_{i-1}\alpha\beta$, if follows by Corollary 2.2 that there is an execution γ' starting from $C_{i-1}\alpha\beta$ such that γ and γ' are indistiguishable to p. Then p returns t as the result of its last GetTS in γ'. Since

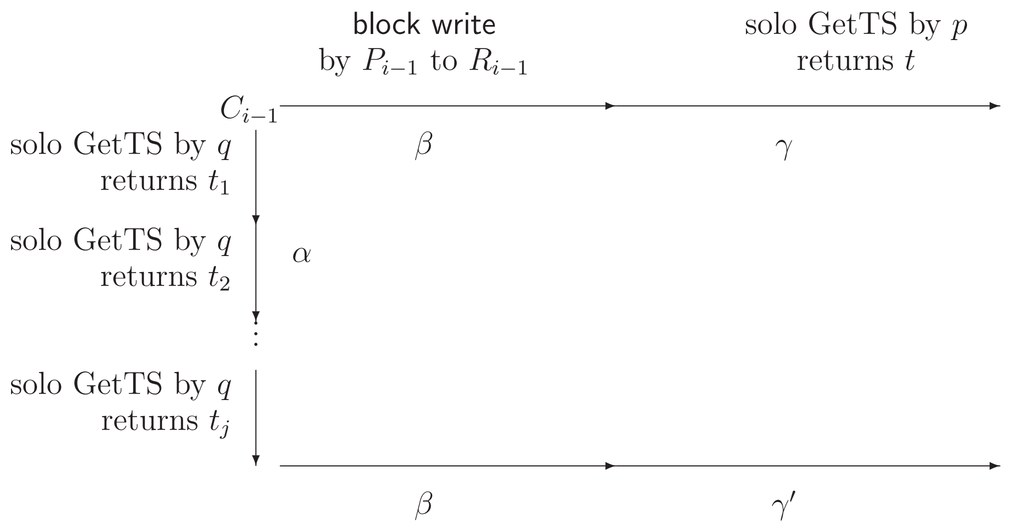


Figure 6.4. Executions used in the proof of Theorem 6.4.

the j'th instance of GetTS by q in α finishes before the last instance of GetTS by p in γ' begins, $t_j < t$. This is a contradiction. Hence q must write to some register outside R_{i-1} in its solo execution starting from C_{i-1}.

Let α' be the prefix of α after which q first covers some register $R \notin R_{i-1}$. Let C_i be the resulting configuration. Let $P_i = P_{i-1} \cup \{q\}$ and let $R_i = R_{i-1} \cup \{R\}$. Thus, the claim is true for i and, hence, for $i = 0, \ldots, n$. □

The fact that the universe of possible timestamps is $\mathbb{N}$ under its usual ordering is only used to argue that there are only a finite number of timestamps less than t. The proof we presented actually holds when $(U, <)$ is any *nowhere dense* partially ordered set. This means that, for every two elements $x, y \in U$, there are only a finite number of elements $z \in U$ such that $x < z < y$. Some examples of nowhere dense partially ordered sets are $\mathbb{N}$ under its usual ordering, the set of all finite sets, ordered by set inclusion, and the set of integer vectors of length $k \geq 2$, under component-wise ordering, i.e. $x \leq y$ if and only if $x_i \leq y_i$ for all components i. Some examples of partially ordered sets that are NOT nowhere dense are $\mathbb{Q}$ under its usual ordering, $\mathbb{R}$ under its usual ordering, and the set of integer vectors of length $k \geq 2$, ordered lexicographically.

Theorem 6.5. *Any implementation of* GetTS *for n processes, with a nowhere dense partially ordered range, that satisfies solo termination requires at least n registers.*

Lamport, in his paper, *A new solution of Dijkstra's concurrent programming problem*, CACM, volume 17, 1974, pages 453–455, gives a timestamp system with range $\mathbb{N}$ that uses n single-writer registers: Each process p_i stores its current timestamp in its single-writer register R_i and gets a new timestamp by reading all of the single-writer registers and adding one to the largest timestamp that it saw. Initially each register contains 0. The code for process p_i is:

```
$t \leftarrow 1 + \max\{R_0, \ldots, R_{n-1}\}$
$R_i \leftarrow t$
return $t$
```

Theorem 6.5 shows that this implementation is optimal, even if multi-writer registers are available, provided the universe of possible timestamps is a nowhere dense partially ordered set. Is it possible to use fewer registers when the universe can be more general?

The following algorithm uses $n-1$ single writer registers, instead of n. Its timestamps are pairs of integers, ordered lexicographically. The code for processes $p_0, \ldots, p_{n-2}$ is the same as in Lamport's algorithm, except that they return $(t, 0)$ instead of t. The code for process p_{n-1} is:

```
$t \leftarrow \max\{R_0, \ldots, R_{n-2}\}$
if $t > old$ then $c \leftarrow 1$
            else $c \leftarrow c + 1$
$old \leftarrow t$
return $(t, c)$
```

The code involves two persistent variables, *old* and c, which are both initially 0. The idea for this simple algorithm came from understanding the lower bound. It is not a particularly useful algorithm, but it does show the necessity of the assumption that the partially ordered set is nowhere dense.

Without the restriction that the set of possible timestamps is nowhere dense, it is still possible to get a linear lower bound on the number of registers, but the proof is more involved.

Consider any timestamp implementation from registers that satisfies solo termination. Suppose there is a configuration in which each register in some subset is covered by at least two processes. Then we can show that, from this configuration, it is possible to reach another configuration in which some register not in the subset is covered by an additional process.

Lemma 6.6. *Let $B_1, B_2, Q_1,$ and Q_2 be disjoint sets of processes and let C be a reachable configuration in which B_1 and B_2 each cover the same set of registers R. For $i \in \{1, 2\}$, let β_i be the* block write *performed by B_i starting from configuration C. Then there exists $i \in \{1, 2\}$ such that every Q_i-only execution starting from $C\beta_i$ containing a complete instance of* GetTS *writes to some register not in R.*

Proof. To obtain a contradiction, suppose that, for all $i \in \{1, 2\}$, there is a Q_i-only execution α_i from $C\beta_i$ that contains a complete instance I_i of GetTS, returns the timestamp t_i, and only writes to registers in R. Without loss of generality, we may assume that the executions α_1 and α_2 are finite. Processes in Q_1 take no steps in $\beta_2\alpha_2\beta_1$, so $C\beta_1 \overset{Q_1}{\sim} C\beta_2\alpha_2\beta_1$ The block write β_1 ensures that the contents of all shared registers are the same in these two configurations. Moreover, $C\beta_1 \overset{B_1}{\sim} C\beta_2\alpha_2\beta_1$ since processes in B_1 take no steps in $\beta_2\alpha_2$ and processes learn nothing from performing writes. Then Lemma 2.1 implies that α_1 can be performed starting from $C\beta_2\alpha_2\beta_1$ and $C\beta_1\alpha_1 \overset{B_1 \cup Q_1}{\sim} C\beta_2\alpha_2\beta_1\alpha_1$. Similarly, α_2 can be performed starting from $C\beta_1\alpha_1\beta_2$ and $C\beta_2\alpha_2 \overset{B_2 \cup Q_2}{\sim} C\beta_1\alpha_1\beta_2\alpha_2$.

Since only processes in $B_1 \cup Q_1$ take steps in $\beta_1\alpha_1$ and only processes in $B_2 \cup Q_2$ take steps in $\beta_2\alpha_2$, the executions $C\beta_1\alpha_1\beta_2\alpha_2$ and $C\beta_2\alpha_2\beta_1\alpha_1$ starting from C are indistinguishable to all processes. In other words, starting from configuration C, the executions $\beta_1\alpha_1$ and $\beta_2\alpha_2$ can be performed in either order without changing the resulting state of any process.

This is a problem because each of these executions contains a complete GetTS operation and the results of these two operations must indicate which was performed earlier. Specifically, I_1 is completed before I_2 begins in $\beta_1\alpha_1\beta_2\alpha_2$, so $t_1 < t_2$. Similarly, $t_2 < t_1$, since I_2 is completed before I_1 begins in $\beta_2\alpha_2\beta_1\alpha_1$. This is a contradiction. □

We say that a configuration is *quiescent* if no process has started an instance of an operation that it has not yet finished. This is analogous to the definition of quiescent used in Section 6.1. Suppose that, from every quiescent configuration, it is possible to reach a configuration in which k processes cover registers, but no register is covered by more than three processes. Then, using an argument similar to part of the proof of Lemma 6.2, we show that there is an execution containing arbitrarily many configurations in which k processes cover registers and no register is covered by more than three processes. It follows that it is possible to find two such configurations which are the same in terms of the number of processes covering each register.

Lemma 6.7. *Let Q be a set of processes. Suppose that, from every reachable quiescent configuration, there is a Q-only execution that results in a configuration in which exactly k processes cover registers and no register is covered by more than three processes. Then, from any reachable quiescent configuration D, there is a Q-only history $\alpha\beta_1\beta_2\beta_3\gamma$ such that*

- *there are exactly k processes covering registers in $D\alpha$,*
- *no register is covered by more than three processes in $D\alpha$,*
- *β_1, β_2, and β_3 are* block writes *by disjoint sets of processes to the set of registers that are covered by three processes in $D\alpha$, and*
- *each register is covered by the same number of processes in $D\alpha\beta_1\beta_2\beta_3\gamma$ as it is in $D\alpha$.*

Proof. We inductively define a sequence $E_0, E_1, \ldots$ of configurations reachable from D. Since D is reachable and quiescent, there is a Q-only execution starting from D that results in a configuration E_0 in which exactly k processes cover registers and no register is covered by more than three processes. Let δ_0 be the history of this execution.

For $i \geq 0$, let R_i denote the set of registers that are covered exactly three times in E_i and let $\beta_{1,i}, \beta_{2,i}$ and $\beta_{3,i}$ denote block writes to R_i starting from E_i by disjoint sets of processes. Let ρ_i denote an execution starting from $E_i\beta_{1,i}\beta_{2,i}\beta_{3,i}$ in which each process in Q with a pending operation completes that operation, but no new operations are begun. Then $E_i\beta_{1,i}\beta_{2,i}\beta_{3,i}\rho_i$ is a quiescent configuration. Hence, there is a Q-only execution δ_{i+1} starting from this configuration that results in a configuration E_{i+1} in which exactly k processes cover registers and no register is covered by more than three processes.

There are a finite number of registers and a finite number of processes. So, by the pigeon hole principle, there exist $0 \leq i < j$ such that each register is covered by the same number of processes in E_i as it is in E_j. Let

$$\begin{aligned}
\alpha &= \delta_0\beta_{1,0}\beta_{2,0}\beta_{3,0}\rho_1\delta_1\beta_{1,1}\beta_{2,1}\beta_{3,1}\rho_2\cdots\delta_i,\\
\beta_1 &= \beta_{1,i},\\
\beta_2 &= \beta_{2,i},\\
\beta_3 &= \beta_{3,i}, \text{ and}\\
\gamma &= \rho_i\delta_{i+1}\beta_{1,i+1}\beta_{2,i+1}\beta_{3,i+1}\rho_{i+1}\delta_{i+2}\beta_{1,i+2}\beta_{2,i+2}\beta_{3,i+2}\rho_{i+2}\cdots\delta_j.
\end{aligned}$$

Then each register is covered by the same number of processes in $D\alpha\beta_1\beta_2\beta_3\gamma = E_j$ as it is in $D\alpha = E_i$, there are exactly k processes covering registers in $D\alpha$, no register is covered by more than three processes in $D\alpha$, R_i is the set of registers covered by three processes in $D\alpha$, and β_1, β_2, and β_3 are block writes to R_i by disjoint sets of processes. □

Starting from a reachable configuration in which no register is covered by more than three registers, we use Lemmas 6.6 and 6.7 to obtain another configuration in which no register is covered by more than three registers, but the number of processes covering registers has increased. This allows us to obtain the desired lower bound.

Theorem 6.8. *Any n-process implementation of a timestamp from registers that satisfies solo-termination requires at least* $\lceil (n-1)/6 \rceil$ *registers.*

Proof. We prove inductively that, for $0 \leq k \leq \lfloor n/2 \rfloor$ and from any reachable quiescent configuration D, there is a $\{p_0, \ldots, p_{2k-1}\}$-only history that results in a configuration in which exactly k processes cover registers and no register is covered by more than three processes. Then, for $k = \lfloor n/2 \rfloor$, there are at least $\lceil k/3 \rceil = \lceil (n-1)/6 \rceil$ covered registers.

For $k = 0$, the claim follows with σ_0 being the empty history starting from D.

Let $0 < k \leq \lfloor n/2 \rfloor$ and let D be a reachable quiescent configuration. By the induction hypothesis, from every reachable quiescent configuration, there is a $\{p_0, \ldots, p_{2k-3}\}$-only history that results in a configuration in which exactly $k-1$ processes cover registers and no register is covered by more than three processes. Then, Lemma 6.7 implies that there is a $\{p_0, \ldots, p_{2k-3}\}$-only execution $\alpha\beta_1\beta_2\beta_3\gamma$ starting from D such that there are exactly $k-1$ processes covering registers in $D\alpha$, no register is covered by more than three processes in $D\alpha$, β_1, β_2, and β_3 are block writes by disjoint sets of processes to the set of registers R that are covered by three processes in $D\alpha$, and each register is covered by the same number of processes in $D\alpha\beta_1\beta_2\beta_3\gamma$ as it is in $D\alpha$.

For $i \in \{1, 2\}$, let δ_i be a p_{2k-i}-only history starting from $D\alpha\beta_i$ in which p_{2k-i} performs a complete instance of GetTS. By Lemma 6.6, there exists $i \in \{1, 2\}$ such that p_{2k-i} writes to some register not in R during δ_i. Let λ be the longest prefix of δ_i in which p_{2k-i} only writes to registers in R and let $r \notin R$ be the register that p_{2k-i} covers in $D\alpha\beta_i\lambda$.

Note that $D\alpha\beta_1\beta_2\beta_3$ and $D\alpha\beta_2\beta_1\beta_3$ are indistinguishable to all processes and they are indistinguishable from $D\alpha\beta_i\lambda\beta_{3-i}\beta_3$ to all processes except p_{2k-i}. Moreover, the block write β_3 overwrites all registers written during β_1, β_2, and λ, so each register has the same value in all three of these configurations.

Since γ is a $\{p_0, \ldots, p_{2k-3}\}$-only history Lemma 2.1 implies that γ can also occur starting from $D\alpha\beta_2\beta_1\beta_3$ and $D\alpha\beta_i\lambda\beta_{3-i}\beta_3$ and the resulting configurations, $D\alpha\beta_1\beta_2\beta_3\gamma$, $D\alpha\beta_2\beta_1\beta_3\gamma$, and $D\alpha\beta_i\lambda\beta_{3-i}\beta_3\gamma$, are indistinguishable to all processes except p_{2k-i}. Thus, each process other than p_{2k-i} covers the same register in these configurations and each register other than r is covered by at most three processes.

Register $r \notin R$ is covered by at most two processes in $D\alpha$ and, hence, in $D\alpha\beta_1\beta_2\beta_3\gamma$. Therefore, it is covered by at most three processes in $D\alpha\beta_i\lambda\beta_{3-i}\beta_3\gamma$.

Finally, there are exactly $k-1$ processes in $\{p_0, \ldots, p_{2k-3}\}$ that cover registers in $D\alpha$ and, hence, in $D\alpha\beta_1\beta_2\beta_3\gamma$ and in $D\alpha\beta_i\lambda\beta_{3-i}\beta_3\gamma$. Thus, including p_{2k-i}, there are exactly k processes that cover registers in $D\alpha\beta_i\lambda\beta_{3-i}\beta_3\gamma$. This proves the claim for k. ☐

Theorem 6.4, Theorem 6.5, the second algorithm, and Lemma 6.6 are from *The Space Complexity of Unbounded Timestamps*, by Faith Ellen, Panagiota Fatourou, and Eric Ruppert, which appears in Distributed Computing, volume 21, 2008, pages 103–115. Lemma 6.7 and Theorem 6.8 are from *The Space Complexity of Long-lived and One-Shot Timestamp Implementations*, by Maryam Helmi, Lisa Higham, Eduardo Pacheco, and Philipp Woelfel, which appears in PODC 2011, pages 139–148.

6.3 Space and Step Complexity Lower Bounds for the Implementation of a Counter using Swap Objects

Next, we consider the problem of implementing a *counter*, an atomic object whose set of values are the nonnegative integers and which supports two operations: READ, which returns the current value of the object, and INCREMENT, which increases the value of the object by 1. The initial value of a counter is 0.

The model we consider is asynchronous shared memory system in which n processes communicate using swap objects that support the swap and read primitives.

We will prove the following lower bound:

Theorem 6.9. *Any n-process implementation of a counter using only swap objects requires at least $n-1$ swap objects and, in the worst case, a READ takes at least $n-1$ steps.*

Fix any implementation of a counter and fix a process p. In this proof, an adversary will construct an execution (starting from an inititial configuration) in which p accesses $n-1$ different swap objects while performing a READ. It will inductively construct a sequence of histories $\alpha_k\beta_k\pi_k$, for $k = 0, \ldots, n-1$ where

- $\alpha_k\beta_k$ is by set of k processes, not including p, in which each is performing INCREMENT,
- β_k is a block swap of a set H_k of k swap objects, and
- π_k is the history of a prefix of a solo execution of a READ by p, in which H_k is the set of swap objects it accesses.

The base case, $k = 0$ is easy: Let α_0, β_0, and π_0, be empty histories and let H_k be the empty set of swap objects. So, let $0 \le k < n-1$. Suppose the history $\alpha_k\beta_k\pi_k$ has

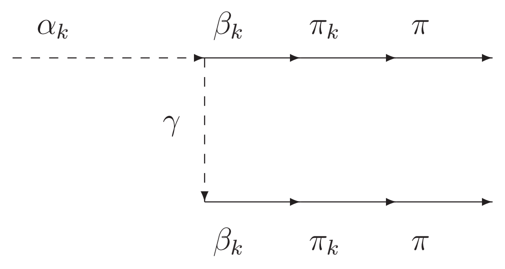


Figure 6.5. The histories used in the proof of Claim 6.10.

been constructed. Let π be the extension of π_k in which p finishes the solo execution of its READ. Only $k < n-1$ processes take steps in $\alpha_k\beta_k$, so there is a process $q \neq p$ that takes no steps in $\alpha_k\beta_k$. Let γ be the history of a solo execution by q starting immediately after α_k, in which it performs $k+1$ complete INCREMENTS.

Claim 6.10. *During $\pi_k\pi$, process p must access a swap object not in H_k that was modified by q.*

Proof. Suppose not. Then the executions with histories $\alpha_k\gamma\beta_k\pi_k\pi$ and $\alpha_k\beta_k\pi_k\pi$ are indistinguishable to p. This is because all the swap objects in H_k have the same values after the block swap, β_k, in both executions and, hence, all the swap objects that p accesses during $\pi_k\pi$ have the same values in both executions. It follows that p returns the same value for its READ in both executions.

Let c be the number of INCREMENTS completed in α_k. In β_k, one step is performed by each of k different processes, so there are at most k additional complete or incomplete INCREMENTS in $\alpha_k\beta_k$. Thus, the value that p returns for its READ in $\alpha_k\beta_k\pi_k\pi$ is at most $c+k$. In $\alpha_k\gamma$, there are $c+k+1$ completed INCREMENTS, so the value that p returns for its READ in $\alpha_k\gamma\beta_k\pi_k\pi$ is at least $c+k+1$. Therefore p returns different values for its READ in these two executions. This is a contradiction. □

Let π_{k+1} be the shortest prefix of $\pi_k\pi$ in which p accesses a swap object H not in H_k and let $H_{k+1} = H_k \cup \{H\}$. Then π_{k+1} is the history of a prefix of a solo execution of a READ by by p in which H_{k+1} is the set of swap objects it accesses. Note that π_k is a proper prefix of π_{k+1}, since p only accesses swap objects in H_k during π_k.

Let ψ be the first access of H by q in γ and let γ' be the prefix of γ up to, but not including ψ. Let $\alpha_{k+1} = \alpha_k\gamma'$ and let $\beta_{k+1} = \psi\beta_k$. Then $\alpha_{k+1}\beta_{k+1}$ is the history of an execution by a set of $k+1$ processes, not including p, in which they are performing INCREMENTS, β_{k+1} is a block swap of H_{k+1}, and the claim holds for $k+1$. Hence, by induction, the claim holds for all k such that $0 \leq k \leq n-1$.

Unfortunately, there is a mistake in this proof. The problem is that the first swap object $H \notin H_k$ that p accesses in π might not be a swap object that q accesses. For example, p might access two swap objects not in H_k, but only the second is also accessed by q.

An alternative is to define H to be the first swap object that q accesses which is not in H_k. In this case, we can get an even simpler construction. Specifically, the history will be $\alpha_k\beta_k$, where

- $\alpha_k\beta_k$ is by a set of k processes, in which each is performing INCREMENT, and

- β_k is a block swap of a set H_k of k swap objects.

Then, in the inductive step, it suffices to choose any distinct processes p and q that take no steps in $\alpha_k\beta_k$ and to let π be the history of a solo execution by p of a READ starting immediately after $\alpha_k\beta_k$. The proof of the claim and the definitions of α_k and β_{k+1} remain the same. However, from this proof, we only get a lower bound of $n-1$ on the number of swap objects needed by the implementation, but not on the number of steps taken by a READ.

Instead, define H to be the first swap object that q accesses which is not in H_k and which is accessed by p in $\pi_k\pi$. Then let π_{k+1} be the shortest prefix of $\pi_k\pi$ in which p accesses $H \notin H_k$. The problem now is that p may access other objects not in H_k before it accesses H. So, it is not necessarily the case that $H_{k+1} = H_k \cup \{H\}$ is the set of swap objects that p accesses in π_{k+1}. However, the equality is not important. It suffices that we inductively maintain that H_k is a *subset* of the processes that p accesses in π_k. Specifically, the inductively constructed histories are $\alpha_k\beta_k\pi_k$, where

- $\alpha_k\beta_k$ is an execution by a set of k processes, not including p, in which each is performing INCREMENT,
- β_k is a block swap of a set H_k of k swap objects, and
- π_k is the history of a prefix of a solo execution of a READ by p, in which it accesses every swap object in H_k.

The proof of Claim 6.10 still holds.

Finally, we have to be careful about the definition of π_{k+1}. It should *not* be the shortest prefix of $\pi_k\pi$ in which p accesses H. The problem is that p might access H near the beginning of π_k and, so, p might not access all the swap objects in H_k during π_{k+1}. A good way to define π_{k+1} is as the shortest prefix of $\pi_k\pi$ in which p accesses every swap object in H_{k+1}. Then the claim holds for $k+1$.

This lower bound was first proved by Prasad Jayanti, King Tan, and Sam Toueg in their paper *Time and Space Lower Bounds for Nonblocking Implementations*, which appeared in SIAM Journal on Computing, volume 30, number 2, 2000, pages 438-456.

6.4 A Lower Bound on Step Complexity for Space-Optimal Implementations of a Multi-Writer Snapshot

We consider wait-free implementations of an m-component snapshot object shared by $n > m$ processes, each of which can update any component. Our model is asynchronous shared memory, where processes communicate through multi-writer registers.

Fix any implementation of an m-component snapshot object shared by $n > m$ processes that communicate using at most m registers. We will prove that it uses at least m registers and, in the worst case, a SCAN takes $\Omega(mn)$ steps. These lower bounds are tight.

We begin with the definition of *fatal configuration*. It is fatal in the sense that, if an implementation ever reaches a fatal configuration, it is incorrect. Using this definition,

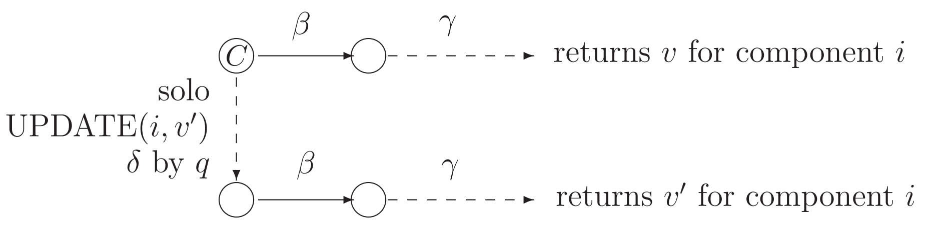


Figure 6.6. The executions in the proof of Lemma 6.11.

we will prove the space lower bound and a number of structural properties of space optimal implementations. Then we will inductively construct a long SCAN.

A configuration C is *k-fatal* if there exists a set, P, of $k \leq m < n$ processes such that, in C, processes in P cover k different registers, and processes in P are performing UPDATES to fewer than k different components. A configuration is *fatal* if it is k-fatal for some $k \geq 1$.

Lemma 6.11. *No execution can reach a fatal configuration.*

Proof. Proof by contradiction. Consider the largest value of k for which there is a reachable k-fatal configuration. In a k-fatal configuration, C, there is a set P of k processes that covers a set R of k registers and the processes in P are performing UPDATES to a set I of fewer than k components. Without loss of generality, we may assume that all processes not in P are idle, because we can simply let each process not in P perform steps until it completes its operation.

Consider the execution starting from C in which the processes in P perform a block write β and then some process $p \in P$ performs a solo execution, γ, in which it completes its current operation and then performs a SCAN. Pick any component $i \notin I$ and let v be the value of component i in the response from this SCAN.

Pick a process $q \notin P$. There is such a process since there are n processes and $n > m \geq k$. Let δ be the solo execution by q starting from C in which it UPDATES component i to a new value $v' \neq v$.

Process q will eventually write to a register $r \notin R$. To see why, suppose that q only writes to registers in R during δ. Then the executions $\beta\gamma$ and $\delta\beta\gamma$ starting from C are indistinguishable to p. Hence, p must return the same response from its SCAN in both executions. In particular, it must return value v for component i in $\beta\gamma$. However, since the UPDATE by process q finishes before the SCAN by p begins in $\delta\beta\gamma$, and there are no pending UPDATES to component i in configuration C, process p must return value v' for component i in $\delta\beta\gamma$ starting from C. This is a contradiction.

Thus, there is a prefix δ' of δ such that q covers a register $r \notin R$ in configuration $C\delta'$. This configuration is $(k+1)$-fatal. This contradicts the maximality of k. □

Now we present the structural lemmas.

Lemma 6.12. *Processes performing SCAN operations do not write to registers.*

Proof. Otherwise, there is a configuration in which some process is performing a SCAN and is covering a register. This configuration is 1-fatal. □

Lemma 6.13. *Processes performing solo executions of UPDATE operations to the same component starting from the same configuration, first write to the same register.*

Proof. Suppose there are solo executions of UPDATE operations by p and q to the same component starting from the same configuration, C, that first write to different registers. Run p and q starting from C until just before their first writes. The resulting configuration is 2-fatal. □

Lemma 6.14. *A process performing a solo execution of an UPDATE operation to a particular component, i, starting from some configuration, C, first writes to the same register, no matter what new value it is using for its UPDATE.*

Proof. Suppose that process p first writes to two different registers, R and R', when performing solo executions of UPDATE(i, v) and UPDATE(i, v'), respectively, starting from configuration C.

Let γ be a solo execution by some other process q starting from C in which it finishes its pending operation, if any, and then performs an UPDATE to component i until it first covers a register R''. Without loss of generality, suppose $R'' \neq R$. Let α denote the longest prefix of p's solo execution of UPDATE(i, v) starting from C in which p does not write.

Since $C\alpha$ is indistinguishable from C to process q, γ is an execution starting from $C\alpha$. However, the resulting configuration $C\alpha\gamma$ is 2-fatal. □

Let C_0 be the initial configuration in which all components have value $\perp$ and let R_i be the register to which a process first writes when performing a solo execution of UPDATE to component i starting from configuration C_0.

Lemma 6.15. *Let α be an execution starting from C_0 in which some process takes no steps. Then all UPDATE operations to component i in α only write to R_i.*

Proof. Suppose that process p writes to a register $R \neq R_i$ during an UPDATE operation to component i. Let α' denote a prefix of α such that, at the end of α', p is covering R while performing an UPDATE to component i.

Let q be a process that takes no steps in α. By Lemma 6.13, a solo execution of an UPDATE to component i by q starting from C_0 first writes to R_i. Let β be the longest prefix of this solo execution in which q performs no writes.

Since C_0 and $C_0\beta$ are indistinguishable to process p, $\beta\alpha'$ is an execution starting from C_0. However, the resulting configuration, $C_0\beta\alpha'$ is 2-fatal. □

Lemma 6.16. *$R_i \neq R_j$ for $i \neq j$.*

Proof. Suppose there are solo executions UPDATE(i, v) by process p and UPDATE(j, v') by process q starting from configuration C_0 that both write to the same register R. Note that both p and q are idle at C_0, so by Lemma 6.15, they only write to register R during these executions.

Let α denote the prefix of UPDATE(i, v) until process p first covers register R and let α' denote the rest of this solo execution followed by a solo execution of a SCAN by p. Let β denote the solo execution of UPDATE(j, v') by process q. Since C_0 and $C_0\alpha$ are indistinguishable to process q, $\alpha\beta$ is an execution starting from C_0. Since α'

begins with a write to register R and all writes in β are to register R, $\alpha\beta\alpha'$ is also an execution starting from C_0 and is indistinguishable from $\alpha\alpha'$ to p.

The SCAN by p in execution $\alpha\alpha'$ must return value $\perp$ for component j, since $\alpha\alpha'$ contains no UPDATES to component j. The SCAN by p in execution $\alpha\beta\alpha'$ must return value v' for component j, because it starts after the UPDATE to component j by q is finished. This is impossible, since these two executions are indistinguishable to p. □

From this lemma, we immediately get our space lower bound.

Theorem 6.17. *Any implementation of an $m < n$ component snapshot object shared by n processes requires at least m registers.*

Next, we prove our lower bound on the step complexity of SCAN.

Theorem 6.18. *In any space-optimal implementation of an $m < n$ component snapshot object shared by n processes, a process requires $\Omega(mn)$ steps to perform a SCAN in the worst case.*

We first outline the proof of this theorem. An adversary constructs a bad execution where a troublesome SCAN by some process t performs $\Omega(n)$ batches of $m-1$ reads. The adversary ensures that during this time, the snapshot always has at least one component that contains the value 1, which t has never seen. This implies that t cannot have terminated.

The bad execution employs $n-4$ *visible processes*, $p_1, \ldots, p_{n-4}$, each of which has started an UPDATE with value 0 to a carefully chosen component and is covering a register before t starts. To prevent t from seeing the hidden value 1, just before t reads from any register that might contain value 1, the adversary causes one of the visible process to write obsolete information to it. The bad execution also employs two *hidden processes*, q and q', performing UPDATES, so the snapshot object always contains at least one component with value 1, and SCANS, to constrain the linearization points of the UPDATES by themselves and by the visible processes.

For each batch of reads of $m-1$ different registers that t performs, the adversary uses one visible process to overwrite the last of these registers just before t reads it. Thus the adversary can schedule $\Omega(n)$ such batches. There is also one process that performs no steps in the bad execution. This is useful so that we can apply some of the structural lemmas.

Proof. Formally, we construct a sequence of components, $i_1, \ldots, i_{n-4}$, and a sequence of executions, $\alpha_0, \ldots, \alpha_{n-4}$, starting from C_0 such that, for every k, where$0 \leq k \leq n-4$,

- $\alpha_k = \beta_k \cdots \beta_1 \cdot \lambda_1 \cdot w_1 \cdots \lambda_k \cdot w_k$,
- w_j is the first write (to register R_{i_j}) by process p_j when it performs a solo execution of UPDATE$(i_j, 0)$ starting from C_0 and β_j is the portion of this execution preceding w_j,
- $\lambda_1 \cdots \lambda_k$ is the prefix of a single SCAN performed by process t,
- process t reads from all registers except R_{i_j} during λ_j and is about to access R_{i_j} at the end of λ_j, and

- if t runs by itself starting from $C_0\alpha_k$, it will read every register before completing its SCAN.

For the base case, $k = 0$, let α_0 be the empty execution starting from C_0. Let λ be the solo execution of a SCAN by t starting from $C_0\alpha_0 = C_0$. Suppose t doesn't read some register R_ℓ during λ. Let γ be the solo execution of UPDATE$(\ell, 1)$ by process q starting from C_0. By Lemma 6.15, process q only writes to R_ℓ during γ, so $\gamma\lambda$ is an execution that is indistinguishable from λ to process t. Hence, p must return the same value for component ℓ in both these executions. However in λ, process p has to return $\perp$ for component ℓ and in $\gamma\lambda$, process p has to return 1 for component ℓ. This is a contradiction.

Let $1 \leq k \leq n-4$ and suppose the claim is true for $k-1$. By the induction hypothesis, if t runs by itself starting from configuration $C_0\alpha_{k-1}$ will eventually read every register. Let λ_k be the longest solo execution by t starting from $C_0\alpha_{k-1}$ in which t does not read from every register. Let R_{i_k} be the register that t is about to read at the end of λ_k.

Let w_k be the first write (to register R_{i_k}) by process p_k when it performs a solo execution of UPDATE$(i_k, 0)$ starting from C_0 and let β_k be the portion of this execution preceding w_k. Then

$$\begin{aligned}\alpha_k \quad = \quad & \beta_k\beta_{k-1}\cdots\beta_1\cdot \\ & \lambda_1\cdot w_1\cdot \\ & \vdots \\ & \lambda_{k-1}\cdot w_{k-1}\cdot \\ & \lambda_k\cdot w_k\end{aligned}$$

is an execution. Let λ be the solo execution by t starting from $C_0\alpha_k$ until it finishes its SCAN.

It remains to prove that t reads every register during λ. Suppose t doesn't read register R_ℓ during λ. We will construct an execution α'' that is indistinguishable from $\alpha = \alpha_k\lambda$ to t, but in which t must return a different response. Specifically, we will show that in α'', process t has to return the value 1 for some component. However, in α, no UPDATES with value 1 are performed and no component has 1 as its initial value, so t can't return the same response.

We start by inserting $k+1$ UPDATES with value 1 by the hidden process q, to components $i_1, \ldots, i_k, \ell$, into α. They are denoted $U_1(i_1, 1), \ldots, U_k(i_k, 1)$, $U_{k+1}(\ell, 1)$. We also insert k SCANS, $S_1, \ldots, S_k$, by process q into α. Let

$$\begin{aligned}\alpha' \quad = \quad & \beta_k\beta_{k-1}\cdots\beta_1\cdot U_1(i_1,1)\cdot \\ & \lambda_1\cdot U_2(i_2,1)\cdot w_1\cdot S_1\cdot \\ & \vdots \\ & \lambda_{k-1}\cdot U_k(i_k,1)\cdot w_{k-1}\cdot S_{k-1}\cdot \\ & \lambda_k\cdot U_{k+1}(\ell,1)\cdot w_k\cdot S_k\cdot \\ & \lambda.\end{aligned}$$

All reads by the visible processes $p_1, \ldots, p_k$ occur in α' before q takes any steps, so α' and α are indistinguishable to them.

By Lemma 6.12, q doesn't write during its SCANS. By Lemma 6.13, q only writes to R_{i_j} during $U_j(i_j, 1)$, for $j = 1, \ldots, k$. The register R_{i_j} is not read by t during λ_j and the contents of R_{i_j} are overwritten by the write w_j of visible process p_j. Therefore, t never reads any value written to R_{i_j} by q during $U_j(i_j, 1)$. Similarly, R_ℓ is the only register q writes to during $U_{k+1}(\ell, 1)$, and, by assumption, t does not read from R_ℓ during λ. Thus, process t does not read any value that q writes and α' is indistinguishable from α to t.

Next, we insert k UPDATES with value 0 by the other hidden process q', to components $i_1, \ldots, i_k$, into α'. They are denoted $U'_1(i_1, 0), \ldots, U'_k(i_k, 0)$. Let

$$\begin{aligned}
\delta \quad = \quad & \beta_k \beta_{k-1} \cdots \beta_1 \cdot U_1(i_1, 1) \cdot \\
& \lambda_1 \cdot U_2(i_2, 1) \cdot U'_1(i_1, 0) \cdot w_1 \cdot S_1 \cdot \\
& \vdots \\
& \lambda_{k-1} \cdot U_k(i_k, 1) \cdot U'_{k-1}(i_{k-1}, 0) \cdot w_{k-1} \cdot S_{k-1} \cdot \\
& \lambda_k \cdot U_{k+1}(\ell, 1) \cdot U'_k(i_k, 0) \cdot w_k \cdot S_k \cdot \\
& \lambda.
\end{aligned}$$

In δ, the operations by the hidden processes don't overlap with one another, so they must be linearized in the following order:

$$\begin{aligned}
& U_1(i_1, 1), \\
& U_2(i_2, 1), U'_1(i_1, 0), S_1, \\
& U_3(i_3, 1), U'_2(i_2, 0), S_2, \\
& \vdots \\
& U_k(i_k, 1), U'_{k-1}(i_{k-1}, 0), S_{k-1}, \\
& U_{k+1}(\ell, 1), U'_k(i_k, 0), S_k.
\end{aligned}$$

The UPDATES by visible processes all use 0, so even if some of them are linearized between $U'_j(i_j, 0)$ and S_j, the SCAN S_j must return 0 for component i_j in δ, for $j = 1, \ldots, k$.

Now consider the execution

$$\begin{aligned}
\alpha'' \quad = \quad & \beta_k \beta_{k-1} \cdots \beta_1 \cdot U_1(i_1, 1) \cdot \\
& \lambda_1 \cdot U_2(i_2, 1) \cdot U'_1(i_1, 1) \cdot w_1 \cdot S_1 \cdot \\
& \vdots \\
& \lambda_{k-1} \cdot U_k(i_k, 1) \cdot U'_{k-1}(i_{k-1}, 1) \cdot w_{k-1} \cdot S_{k-1} \cdot \\
& \lambda_k \cdot U_{k+1}(\ell, 1) \cdot U'_k(i_k, 1) \cdot w_k \cdot S_k \cdot \\
& \lambda.
\end{aligned}$$

This execution is the same as δ except that the UPDATES by process q' use 1 instead of 0. By Lemma 6.13, q' only writes to R_{i_j} during $U'_j(i_j, 0)$ and $U'_j(i_j, 1)$. But R_{i_j} is overwritten by w_j before R_{i_j} can be read by any other process. Therefore, to all other processes, α'' and δ are indistinguishable from α'. Hence, the SCAN S_j must return 0 for component i_j in α'', for $j = 1, \ldots, k$.

As before, in α'', the operations by the hidden processes are linearized in the order

$$\begin{array}{l} U_1(i_1,1), \\ U_2(i_2,1), U_1'(i_1,1), S_1, \\ U_3(i_3,1), U_2'(i_2,1), S_2, \\ \vdots \\ U_k(i_k,1), U_{k-1}'(i_{k-1},1), S_{k-1}, \\ U_{k+1}(\ell,1), U_k'(i_k,1), S_k. \end{array}$$

There are no UPDATES by hidden processes linearized between $U_j'(i_j,1)$ and S_j. Since S_j returns 0 for component i_j, there must be at least one UPDATE to component i_j by a visible process linearized between them, for $j = 1,\ldots,k$. Since there are k SCANS by q and k UPDATES by visible processes, exactly one of these UPDATES must be linearized between $U_j'(i_j,1)$ and S_j, for $j = 1,\ldots,k$.

Note that $i_j \neq i_{j-1}$. This is because t starts λ_j by reading $R_{i_{j-1}}$ and doesn't read R_{i_j} during λ_j. Similarly, $\ell \neq i_k$. Therefore, in the linearization of the operations in α'', every UPDATE with value 0 is preceded by two UPDATES with value 1 to different components. Thus, after $U_1(i_1,1)$, there will always be a component with value 1 in the resulting linearization. Hence, during α'', the troublesome SCAN by t will have to return the value 1 in some component, no matter where it is linearized.

Since t does not return the value 1 in any component during α, and α and α'' are indistinguishable to t, we have a contradiction. Thus t reads every register during λ and the claim is true for k. □

The space and step complexity lower bounds apply to snapshot implementations from any historyless objects, provided that $m < n-1$. The proof also shows that SCAN has a very particular form in any efficient, space-optimal snapshot implementation.

The results in this section are from *Time Lower Bounds for Implementations of Multi-writer Snapshots* by Faith Ellen, Panagiota Fatourou, and Eric Ruppert, Journal of the ACM, Volume 54, Number 6, December 2007.

When $m = n-1$, Theorem 6.18 gives an $\Omega(n^2)$ lower bound on step complexity of SCAN in any implementation of an $(n-1)$-component snapshot object from $n-1$ registers. In contrast, there is an implementation of an n-component snapshot object using n registers with $O(n)$ step complexity for SCAN. See Hagit Attiya and Ophir Rachman, *Atomic Snapshots in $O(n \log n)$ Operations*, SIAM Journal on Computing, volume 27, number 2, pages 319–340, April 1998 and A. Israeli, A. Shaham, and A. Shirazi, *Linear-time Snapshot Implementations in Unbalanced Systems*, Mathematical Systems Theory, volume 28, number 5, pages 469–486, September/October 1995.

6.5 A Lower Bound on the Number of Stalls Incurred by a Counter Implemented Using Arbitrary Read-Modify-Write Primitives

Here, we consider an asynchronous shared memory model in which processes can apply arbitrary read-modify-write primitives to base objects. A *read-modify-write* primitive has a fixed number of input variables. When it is applied to an object, the value of the object may be changed, based on the current value of the object and the values of these variables. A response is returned to the process that applied the primitive.

The time to perform an operation can be influenced, not only by the number of primitives a process applies to objects, but also by the amount of contention it incurs at objects when other processes access them concurrently. A *stall* is a delay that results from waiting for another process that applies a nontrivial primitive to the same object. For example, suppose that in some configuration, there is some object at which no processes are poised. Let k processes each take steps that result in them being poised to apply a nontrivial primitive to this object. Then let each of them take another step. The i'th process to access the object incurs $i-1$ stalls, for $1 \leq i \leq k$.

A process that accesses a shared counter provided by the hardware can incur $n-1$ stalls if the other processes all want to INCREMENT the counter at the same time. Is it possible to reduce the maximum number of stalls the process incurs by implementing the counter using more shared objects, each shared by fewer processes, or by using more powerful shared objects? The following result says that it is not possible.

Theorem 6.19. *Any implementation of a counter shared by n processes has an execution in which some READ incurs at least $n-1$ stalls.*

To prove this result, an adversarial scheduler constructs a bad execution in which some process p incurs $n-1$ stalls while performing a single READ. This bad execution has the form $\alpha\sigma_1 \cdots \sigma_r$. During α, all processes other than p take steps until they cover base objects that will later be accessed by p. Let $B_1, \ldots, B_r$ be the base objects covered by these processes at the end of α and, for $i = 1, \ldots, r$, let S_i be the set of those processes that cover B_i. Then $S_1 \cup \cdots \cup S_r$ is a disjoint union of all processes except p. During σ_i, process p takes steps until it is about to access B_i, then each process in S_i takes one step in which it accesses B_i, and, finally, p accesses B_i. Hence p incurs one stall in σ_i for each process in S_i. In total, p incurs $n-1$ stalls in this execution.

A k-stall execution is an execution with a history $\alpha\sigma_1 \cdots \sigma_r$ where there are distinct base objects $B_1, \ldots, B_r$ and disjoint sets of processes $S_1, \ldots, S_r$ whose union has size k such that

- p is performing a single READ,
- all other processes only perform INCREMENTS,
- p takes no steps in α,
- at the end of α, each process in S_i covers B_i, for $i = 1, \ldots, r$,
- in σ_i, process p runs by itself until just before it first accesses B_i, then each of the processes in S_i accesses B_i, and, finally, p accesses B_i, and
- in every $(\{p\} \cup S_1 \cup \cdots \cup S_r)$-free extension τ of α by processes performing INCREMENTS, no process performs a nontrivial operation on any base object accessed in $\sigma_1 \cdots \sigma_r$.

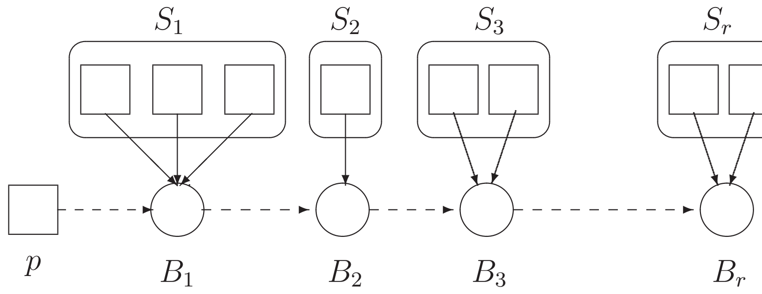


Figure 6.7. The configuration at the end of α in a k-stall execution.

This is illustrated in Figure 6.7. Note that process p incurs k stalls in a k-stall execution. The empty execution is a 0-stall execution.

Using the following lemma, we can inductively construct an $(n-1)$-stall execution. This proves Theorem 6.19.

Lemma 6.20. *If there is a k-stall execution, where $0 \leq k < n-1$, then there is a $(k+m)$-stall execution, for some $m \geq 1$.*

Proof. Suppose there is a k-stall execution with history $\alpha\sigma_1 \cdots \sigma_r$. Let σ be the extension of $\alpha\sigma_1 \cdots \sigma_r$ in which p completes its READ. Let v be the value returned. Let τ be the extension of α in which some process $q \notin \{p\} \cup S_1 \cup \cdots \cup S_r$ performs n complete INCREMENTS.

From the definition of a k-stall execution, during τ, process q performs no nontrivial operations on base objects accessed during $\sigma_1 \cdots \sigma_r$. Thus, by Lemma 2.1, $\sigma_1 \cdots \sigma_r$ can occur following $\alpha\tau$. If we let p finish its READ, then it must return a value larger than v. This is because, at the end of α, there are at most $n-1$ processes with pending INCREMENTS. Thus, the number of INCREMENTS completed by the end of α is at least $v-(n-1)$ and the number of INCREMENTS completed by the end of $\alpha\tau$ is at least $v+1$.

During τ, process q must modify a base object accessed by p during σ; otherwise the executions with histories $\alpha\sigma_1 \cdots \sigma_r\sigma$ and $\alpha\tau\sigma_1 \cdots \sigma_r\sigma$ would be indistinguishable to p.

Consider the set E of all extensions τ' of α by one or more processes not in $\{p\} \cup S_1 \cup \cdots \cup S_r$ which are performing INCREMENTS and such that, at the end of τ', at least one of these processes covers a base object that is accessed by p during σ.

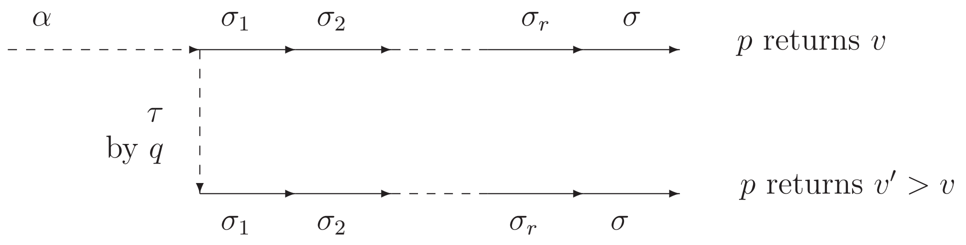


Figure 6.8. The histories in the proof of Lemma 6.20.

The prefix of τ up to, but not including, its first access to a base object accessed by p during σ is an example of a history in E.

Let B_{r+1} be the first base object accessed by p during σ that is covered at the end of one or more of these histories in E. Since $\alpha\sigma_1 \cdots \sigma_r$ is the history of a k-stall execution, no history in E contains a nontrivial operation on any object accessed during $\sigma_1 \cdots \sigma_r$. Hence $B_{r+1} \notin \{B_1, \ldots, B_r\}$. Let σ' be the prefix of σ up to, but not including, p's first access of B_{r+1}.

Among all the histories in E, let τ' be one such that, after $\alpha\tau'$, the largest number of processes cover B_{r+1}. Suppose there are m such processes. Call this set of processes S_{r+1}. Then, when each process in S_{r+1} is scheduled after τ', it first performs a nontrivial operation on B_{r+1}. Let σ_{r+1} denote σ' followed by these m nontrivial operations and then p's access of B_{r+1}. Let $\alpha' = \alpha\tau'$. By the choice of B_{r+1} and the definition of σ', τ' contains no nontrivial operation on any object accessed in σ'. By Lemma 2.1, $\sigma_1 \cdots \sigma_r \sigma_{r+1}$ can occur following α'.

From the definition of B_{r+1} and the maximality of m, it follows that no $(\{p\} \cup S_1 \cup \cdots \cup S_r \cup S_{r+1})$-free extension of α' contains a nontrivial operation on any object accessed in $\sigma_1 \cdots \sigma_r \sigma_{r+1}$. Thus $\alpha'\sigma_1 \cdots \sigma_r \sigma_{r+1}$ is the history of a $(k+m)$-stall execution. $\square$

This result is from Faith Ellen Fich, Danny Hendler, and Nir Shavit, *Linear Lower Bounds on Real-World Implementations of Concurrent Objects*, Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS), October 2005, pages 165–173.