Restricted Stack Implementations

Matei David, Alex Brodsky, Faith Ellen Fich

Department of Computer Science, University of Toronto

Contact
Matei David
Department of Computer Science, University of Toronto
3302—10 King's College Road,
Toronto, Canada
M5S 3G4
`matei@cs.toronto.edu`
+1-416-946-3924

Brief Abstract
We give single-valued Stack (Queue) implementations from Registers and usual consensus number 2 objects, such as Fetch&Add. We also give a Stack implementation for two pushers and any number of poppers. We introduce the BH object type, which captures the computational power of a system of commutative and overwriting consensus number 2 objects, and might prove a useful tool in developing impossibility results.

To be considered for regular track.

Eligible for best student paper award.

Restricted Stack Implementations

Matei David, Alex Brodsky, Faith Ellen Fich

Department of Computer Science, University of Toronto,
10 King's College Road,
Toronto, Canada
`matei|abrodsky|fich@cs.toronto.edu`

Abstract. This paper shows that Stacks (and Queues) shared by any number of processes, but, in which, all stored elements are the same, can be implemented using only commutative and overwriting objects. These include Registers and simple objects of consensus number 2, such as Fetch&Add. It also shows the same result for Stacks that can contain arbitrary values and which support any number of poppers, but at most two pushers.
For these implementations, a new object, BH, is introduced. We prove that a system with one BH object and single-writer Registers has the same computational power as a system with countably many commutative and overwriting objects. This provides a simple characterization of the class of objects that can be implemented from commutative and overwriting objects, and creates a potential tool for proving impossibility results.

1 Introduction

Even though Stacks and Queues are important and well studied data structures, they are not usually available in the hardware and, to use them, one has to implement them from the available basic types. We know Stacks and Queues have consensus number 2 [Her91], so if the distributed system provides object types with consensus number ∞ (Compare&Swap, LL/SC), wait-free Stack and Queue implementations exist, regardless of the number of processes in the system.

In this paper, we consider the question of implementing wait-free Stacks and Queues in systems where only common consensus number 2 types (Fetch&Add, Swap) are available. By Herlihy's universality results [Her91], Stack and Queue implementations exist in systems with two processes, from any consensus number 2 type. In contrast, no such implementations are known when the number of processes is at least 3 and, in fact, it is conjectured that they do not exist [Li01,Dav04b]. Proving this negative result would also solve Herlihy's long-standing open question regarding the ability of Fetch&Add objects to implement every other consensus number 2 type. Working towards settling the conjecture, we give several restricted Stack and Queue implementations.

Since modern distributed systems do provide more powerful types, our results are mainly of theoretical interest. Their relevance stems from the fact that they are dealing with questions at the foundations of our understanding of shared memory distributed computing.

Two operations *commute* if the order in which they are applied does not change the resulting state of the object. One operation *overwrites* another if applying this operation results in the same object state whether or not the other operation is applied immediately before it. Commutative and overwriting objects are those for which every pair of operations performed by different processes either commute or one overwrites the other. Many such objects, including Test&Set objects, Fetch&Increment objects, Fetch&Add objects, Swap objects and Registers are provided in real systems. All of them have consensus number at most 2 [Her91]. The class of all commutative and overwriting read-modify-write objects with consensus number 2 is called Common2.

Using Herlihy's universal construction [Her91], any object of consensus number 2 shared by two processes can be implemented from Registers and Common2 objects. Afek, Weisberger, Weisman [AWW93] prove that any Common2 object shared by any number of processes can be implemented from Registers and any type of objects of consensus number 2. Hence, if a Queue or Stack can be implemented from Registrers and Common2 objects, it can be implemented from Registers and any type of objects of consensus number 2. However, it is conjectured that this is impossible [Li01,Dav04b]. This negative result would imply that the characterization of an object as having consensus number 2 is not sufficient to describe its computational power in systems of more than 2 processes.

Attempts to prove this conjecture for Queues have resulted in the development of a number of restricted implementations of Queues from commutative and overwriting objects (such as Fetch&Add objects, Swap objects, and Registers). Specifically, there are wait-free implementations of Queues shared by one or two dequeuers and any number of enqueuers [HW90,Li01] and wait-free implementations of Queues shared by one enqueuer and any number of dequeuers [Dav04a].

Another natural restriction is to consider Stacks and Queues with domain size 1, i.e. where all the elements stored in the Stack or Queue are the same. Note that single-valued Stacks and Queues behave identically. Push and Enqueue increase the number of stored elements by one. Pop and Dequeue decrease the number of stored elements by one, if there was at least one, and return whether or not this was the case. We prove that a single-valued Stack shared by any number of pushers and poppers can be implemented from commutative and overwriting objects. This means that the difficulty of implementing a general stack is not simply coordinating pushers and poppers so that they can all complete their operations, but must involve poppers determining the order in which the steps of different pushers are linearized.

We obtain our implementation by first constructing an implementation of a Stack with arbitrary domain shared by one pusher and any number of poppers. Then we show how to transform it to obtain an implementation of a single-valued Stack shared by any number of pushers and poppers. We also show how to extend the number of pushers from one to two when the domain is arbitrary. In contrast, it is not known how to implement a Queue shared by two or more enqueuers and any number of dequeuers from commutative and overwriting objects.

The implementations in this paper do not directly use Common2 objects. Instead, we introduce a new object, BH, with a single operation, Sign, and we show that, for any number of processes, the BH object can be implemented from a single Fetch&Add

object. Then we implement our Stacks from a single BH object and one single-writer Register per pusher. The form of our implementations is very simple: To perform a Push, a process appends its current state to its single-writer Register, and performs one or two Sign operations (depending on the implementation). To perform a Pop, a process may perform an Append, followed by two Sign operations and then it collects the single-writer Registers of all pushers.

We also show that any countably infinite collection of Fetch&Add objects and single-writer Registers can be simulated using one BH object and one single-writer Register per process. In this case, a process can perform any operation by appending the operation and its arguments to its single-writer Register, applying one Sign operation to the BH object, and then reading the single-writer Registers of all other processes. Thus, a system with one BH object and one single-writer Register per process and a system with Common2 objects and Registers are *equally powerful*. In particular, to show that an object cannot be implemented from Registers and objects in Common2, it suffices to prove that it has no implementation from one BH object and one single-writer Register per process. Moreover, it suffices to prove the lower bound for a restricted class of implementations in which each operation is simulated by an algorithm with a fixed, very simple form. This restriction enables us to better understand the flow of information between processes and to analyze the interaction between them.

Although the BH object is not an object one would want to implement in hardware or use in an efficient implementation, we believe it is a very useful theoretical tool for studying the computational power of consensus number 2 objects that can be implemented from Registers and objects in Common2, as it provides a simple characterization of the information a process can obtain from such objects during the course of a computation. It has certainly helped us understand why Stacks and Queues are difficult to implement.

2 The BH object type

2.1 BH type definition

Consider an object with only one operation, in which a process appends its own id to a shared log. We refer to an occurrence of a process id in the log as a *signature*. We assume process ids are positive integers, so a list of signatures is a finite sequence of positive integers. The object keeps, by means of its internal state, a complete ordered list of signatures. As a process signs the log, that process receives in response the entire list of signatures, including the one being applied by its current operation. It is not hard to see that such an object has consensus number ∞, so it does not represent the limited power of a system with Registers and Common2 objects.

Informally, a BH object, short for *Blurred History*, works much like the object described above, but it is restricted so that it can be implemented from Registers and Common2 objects. As before, the object has one operation, Sign, and the state of the BH object is the complete list of signatures applied so far. However, the response P_a gets from Sign is not the exact state σ of the BH object, but instead, a set of sequences *indistinguishable* (in the sense defined below) to P_a from σ.

Two sequences of integers σ, σ' are *a-indistinguishable* if there exist $b \neq c$, both different from a, such that $\sigma = \sigma_1 \cdot bc \cdot \sigma_2$, $\sigma' = \sigma_1 \cdot cb \cdot \sigma_2$ and neither b nor c appears in σ_2. In other words, σ' is exactly the same as σ, except for the last consecutive occurences of two different elements other than a, which are swapped. Two sequences σ, σ' are also a-indistinguishable if there is a sequence σ'' which is a-indistinguishable from both. Thus, a-indistinghuishability is transitively closed. We use the terms a-indistinguishable and indistinguishable to P_a to refer to the same realtion.

To provide further intuition, we also give a direct, yet equivalent, defintion for the notion of indistinguishability. We can view a state σ as encapsulating two types of information:

– the *number of signatures* by every process, and
– a *total order* on these signatures (i.e. for each signature, which signatures precede it and which signatures follow it).

Let σ be the state of the BH object immediately after a Sign by P_a. Consider the locations within σ of the last signatures by every process other than P_a. In response to its signing operation, P_a will retrieve the following information from σ:

– the number of signatures by every process, and
– the relative position in the total order of any two signatures, at least one of which is not the last signature by a process other than P_a (i.e. for each signature, except for the last signatures of other processes, which signatures precede it and which signatures follow it).

Hence, P_a won't be able to tell the *relative order* of the last signatures by other processes, when those signatures are consecutive. For example, if the BH object is in state 123 and P_1 applies Sign, the response will be $\{1231, 1321\}$, which we can write as $1\{23\}1$. As a more elaborate example, if the BH object is in the state 1324451671718 and P_9 applies Sign, P_9 will get the response $1\{23\}4\{45\}1671\{178\}9$. Notice that in this example, P_9 can derive the exact location of the last (and only) signature by P_6 because it knows the position of the two surrounding signatures (the second by P_1 and the first by P_7).

When two sequences σ, σ' are a-indistinguishable, one is a permutation of the other, and the set of locations of last signatures by processes other than P_a is the same in both. From the response to a Sign operation, P_a can compute the response of any previous Sign operation by some other process P_b, except possibly for the last operation by P_b. To see this, note that P_a has the position of any signature of P_b except possibly the last, and furthermore, later steps (by P_b and by other processes) can only add information about the exact state at the end of P_b's operation. For example, if P_1 receives the response $124\{24\}353151$ to a Sign operation, it can see that the response P_5 got to its first Sign operation is $124\{234\}5$. In this example, P_1 has the position of the first signature by P_3, but it can see that P_5 couldn't have had that information from the response to its first Sign.

2.2 Implementing a BH object

In this section, we informally explain how to implement a BH object from one Fetch&Add object. A more formal description of this implementation appears in [Dav04b]. The initial state of the BH object is an empty sequence, and the initial value held by the Fetch&Add object in our implementation is 0.

We can view the value V stored in the Fetch&Add object as an infinite sequence of bits, $b_0b_1\ldots$. Let N denote the number of processes in the system. Let $V_1,\ldots,V_N$ be N infinite subsequences of bits of V which are mutually disjoint. For example, allocate the bits of V in Round-Robin fashion, so V_a consists of $\{b_j|(j \bmod N)+1=a\}$. At any point in time, V_a encodes a finite sequence of non-negative integers in such a way that any value can be appended at the end of the sequence by only changing certain bits of V_a from 0 to 1 (e.g. u_1,u_2,u_3 can be encoded as $1^{1+u_1}01^{1+u_2}01^{1+u_3}00..$).

We implement every Sign operation by P_a using one Fetch&Add operation on V that appends a number to the sequence encoded in V_a. Since P_a is the only process changing V_a, it can keep V_a in a local register v_a. Whenever P_a needs to append a number to the sequence encoded in V_a, it can inspect v_a to decide which bits of V_a have to be set from 0 to 1. P_a can then set those bits by a Fetch&Add operation on V with an appropriate argument. For example, if V_a stores $2,0,3$, encoded as 111010111100.., and P_a needs to append the value 1 to this sequence, it has to set the 11-th and 12-th bits of V_a from 0 to 1. P_a can achieve this by performing a Fetch&Add operation on V with argument $2^{a-1+11N}+2^{a-1+12N}$.

In our BH implementation, every process P_a has, in addition to v_a, a second local register w_a. The latter is used to store the last value recieved by P_a from a Fetch&Add operation on V. A high-level Sign is implemented as follows:

- P_a computes, using v_a and w_a, a value x such that performing a Fetch&Add operation on V with argument x has the effect of appending w_a to the sequence encoded in V_a;
- P_a performs Fetch&Add on V with argument x;
- P_a stores in w_a the value received as response to its Fetch&Add operation;
- P_a updates v_a to again mirror V_a;
- P_a computes from w_a the response to the high-level Sign operation.

We have already argued that the computation of x is possible, so all we have left to explain is how to compute the return value for the high-level Sign.

From the value retrieved as response to its Fetch&Add operation, P_a can compute the number of previous signatures by some other process P_b as simply the number of elements in the sequence encoded in V_b. Let $u_{b,i}$ be the i-th number in the sequence encoded in V_b. Then $u_{b,i}$ is the value retrieved by P_b as response to its Fetch&Add operation during its $(i-1)$-st high-level Sign operation. Hence, P_a can compute from $u_{b,i}$ the relative position in the total order of the $(i-1)$-st signature by P_b. The only information about the signature log that P_a might not be able to compute is the relative order of the last signatures by some other processes, when those signatures are consecutive. This is precisely the information needed to construct the class of states indistinguishable to P_a from the signature log.

2.3 The power of a BH object

In this section, we show that a system with of one BH object and one single-writer Register per process can be used to simulate a system with infinitely many Common2 objects and Registers. To do that, we implement a countably infinite collection of Fetch&Add objects and SW Registers using one BH object and one SW Register per process. Our claim follows from the fact that any Register can be implemented from SW Registers [HW90], and that any Common2 object can be implemented from Fetch&Add objects and Registers [AWW93].

Consider a system of countably infinitely many Fetch&Add objects and SW Registers. Assume the objects in this system are indexed by positive integers. A process may perform three types of operations: Fetch&Add(k,x), if k is the index of a Fetch&Add object; Read(k) and Write(k,x), if k is the index of a Register. In a system with one BH object and one SW Register per process, we implement each of the three types of operations as follows:

– P_a appends the current high-level operation to its Register;
– P_a signs the BH object;
– P_a collects the Registers of all processes;
– P_a locally computes the result of the implemented operation.

Throughout the implementation, the value held in P_a's Register is a complete ordered list of all the high-level operations started by P_a. We linearize a high-level operation at the moment the process executing it signs the BH object. Thus, given the responses P_a gets to its Sign and Read operations, P_a can compute all the high-level operations that have occured so far. It can also compute the linearization of these operations, except for what is blurred in the response it got from the BH object. This information is enough for P_a to compute the result of its high-level operation:

– If the high-level operation is a Write, its response is simply OK.
– If the high-level operation is Read(k), we know that only one process P_b might have written to that object (recall that we are considering SW Registers). In this case, P_a returns as result the argument of the last Write operation by P_b linearized before this Read.
– If the high-level operation is Fetch&Add(k,x), P_a needs to compute the sum of the arguments of all the Fetch&Add operations on this object linearized before the current one. Note that P_a does not need to know the order in which these opeartions are linearized, since addition is commutative.

Something stronger can be said about a system with one BH object and one SW Register per process.

Theorem 1. *Let $\mathcal{S}_1$ be a system with countably infinitely many Common2 objects and Registers. Let $\mathcal{S}_2$ be a system with one BH object and one SW Register per process. If there exists an implementation of some object O in $\mathcal{S}_1$, then there exists an implementation of O in $\mathcal{S}_2$. Furthermore, the implementation of a high-level operation on O by a process P_a begins with P_a appending this operation to its SW Register and then alternately performing Sign and Reads of all Registers.*

Corollary 1. *If every process can apply only one type of high-level operation on O (with no parameters), there exists an implementation of O from Common2 objects and Registers if and only if there exists an implementation of O from one BH object.*

Although we do not include formal proofs in this extended abstract, we give the two main ideas needed to establish these results. On the one hand, in a deterministic implementation, the next access to a shared object by P_a is completely determined by the interaction between P_a and the shared memory, and by the high-level operations that P_a is applying. On the other hand, as pointed out in Section 2.1, P_a can compute the response obtained by P_b to any previous Sign operation, except possibly for the last operation by P_b. These two facts allow a process to anticipate most of the values written to Registers by other processes.

3 Stack Implementations from a BH object

3.1 A single-pusher Stack implementation

In this section, we give a single-pusher many-popper Stack implementation from one BH object B and an unbounded array V of SW Registers, all written by the pusher, and each capable of holding one element in the Stack. Let P_1 be the (single) pusher, and let P_a be the poppers, for $a > 1$. The state of B is initially the empty sequence.

The pusher P_1 holds a local variable *last*, initialized to 0, which is used to store the index of the last slot of V to which P_1 wrote. To push an element x on the Stack, P_1 increments *last* and writes x into $V[last]$. P_1 then applies a Sign operation on B. We refer to signatures of P_1 in B as push steps.

To pop an element off the Stack, P_a first applies two Sign operations on B. From the result of its second operation, which is an equivalence class of a-indistinguishable sequences, P_a selects any representative σ. P_a then computes the function f on σ. The value obtained from this computation is either 0, in which case P_a reports an empty Stack, or a positive integer, which is the index in V of the value P_a outputs as result of its Pop. We refer to signatures by a popper P_a in B as pop steps. The signature produced by the first Sign within a Pop operation is a first pop step, and the one produced by the second Sign is a second pop step.

The heart of this implementation is the function f, which takes as input a BH state σ, and decides what value the process executing it should pop from the Stack. Inside the function, we consider each Push operation ϕ, starting with the latest, and try to match it with the earliest completed Pop operation α that starts after the push step of ϕ. If no such α exists, we erase ϕ from σ and continue. On the other hand, if α exists, we erase both α and ϕ from σ and continue. If α turns out to be the Pop operation that invoked f on σ, which is the case if the second pop step of α is the last signature in σ, we decide that α should output the value pushed on the Stack by ϕ.

For the purposes of proving the correctness of this implementation, it will be convenient to assume that P_1 is pushing the values $1, 2, 3, \ldots$, thus identifying the value stored in a cell of V with the index of that cell.

A crucial fact in proving the correctness of this algorithm is given in Lemma 5, where we show that the choice of a representative made in line 4 does not affect the output of a Pop operation. In order to establish this result, we prove several Lemmas saying that, under certain conditions, swapping two consecutive steps in σ does not change the result of f. We never try to move the last pop step in σ, as that is the second pop step of the Pop operation invoking f.

Procedure P_1:`Push(x)`

```
1.  increment( last )
2.  Write( V[ last ], x )
3.  Sign( B, 1 )
```

Procedure P_d:`Pop`, for $d > 1$

```
4.  Sign( B, d )
5.  C ⟵ Sign( B, d )
6.  σ ⟵ any sequence in C
7.  l ⟵ f(σ)
8.  if l = 0
9.        return ε
    else
10.       return Read( V[ l ] )
    endif
```

Function `f(σ)`

```
11. while there exist push steps in σ
12.       $i$ ⟵ location of last push step in σ
13.       $A$ ⟵ { ( $j$, $j'$ ) : $j$ and $j'$ are the indices of the first
                and second steps of a pop operation and $i < j$ }
14.       if $A$ is not empty
15.             ( $k$, $k'$ ) ⟵ pair with minimum $j'$ in $A$
16.             if $k'$ is the last location in σ
17.                   return number of push steps in σ
                endif
18.             delete locations $i$, $k$ and $k'$ from σ
          else
19.             delete location $i$ from σ
          endif
    endwhile
20. return 0
```

Fig. 1. A Single-Pusher Implementation

Let $\sigma = \sigma_1 \cdot ab \cdot \sigma_2$ and $\sigma' = \sigma_1 \cdot ba \cdot \sigma_2$, such that $a \neq b$ and σ_2 is not empty. Lemmas 1, 2 and 3 describe situations in which $f(\sigma) = f(\sigma')$.

Lemma 1. *Swapping two consecutive pop steps by different processes, of which at least one is a first pop step, does not affect the result of f. Formally, if a is a first pop step and b is a pop step, then $f(\sigma) = f(\sigma')$.*

Proof. During every iteration of the `while` loop, membership in A is determined in line 13 by the order between push steps and first pop steps, and the selection of a pop operation in line 15 is determined by the order between second pop steps. Hence, the computations of f on σ and σ' take exactly the same decision during every iteration of the `while` loop.

The same argument can be used to show:

Lemma 2. *Swapping a consecutive push step and second pop step does not affect the result of f. Formally, if a is a push step and b is a second pop step, $f(\sigma) = f(\sigma')$.*

Lemma 3. *Swapping two consecutive second pop steps does not affect the result of f. Formally, if both a and b are second pop steps, $f(\sigma) = f(\sigma')$.*

Proof. We use induction on the number of executions of the `while` loop to show that $f(\sigma) = f(\sigma')$.

It is not hard to see that the only difference in the computations of f on σ and σ' might come in an iteration in which both pop operations involved in the swap are in the set A, one of them is selected in $f(\sigma)$ and the other is selected in $f(\sigma')$. Let τ and τ' be the sequences at the beginning of that iteration, respectively. Without loss of generality, we must have

$$\tau = \tau_1, 1, \tau_2, a_1, \tau_3, b_1, \tau_4, \underline{a_2}, \underline{b_2}, \tau_5$$

$$\tau' = \tau_1, 1, \tau_2, a_1, \tau_3, b_1, \tau_4, \underline{b_2}, \underline{a_2}, \tau_5.$$

Here, the 1 following τ_1 is the last step by the pusher P_1, hence $\tau_2, \tau_3, \tau_4, \tau_5$ contain no 1s. The steps a_1 and a_2 are the first and second steps of a pop operation by P_a, and b_1 and b_2 are the first and second steps of a pop operation by P_b. Notice that in this case τ_3 cannot contain pop steps by P_a because P_a has a pending operation.

In this scenario, $1, a_1, a_2$ are deleted in $f(\sigma)$ and $1, b_1, b_2$ are deleted in $f(\sigma')$. Let

$$\bar{\tau} = \tau_1, \tau_2, \tau_3, b_1, \tau_4, b_2, \tau_5$$

$$\bar{\tau}' = \tau_1, \tau_2, a_1, \tau_3, \tau_4, a_2, \tau_5.$$

So $f(\sigma) = f(\bar{\tau})$ and $f(\sigma') = f(\bar{\tau}')$. The computation of f is not affected by what popper is performing a particular Pop operation, only by the indices of those pop steps. Hence, $f(\bar{\tau}) = f(\bar{\tau}'')$, where $\bar{\tau}'' = \tau_1, \tau_2, \tau_3, a_1, \tau_4, a_2, \tau_5$. Since τ_3 contains no push steps or pop steps by P_a, $\bar{\tau}'$ can be transformed into $\bar{\tau}''$ by a series of swaps of consecutive pop steps, of which one is the first pop step a_1. By Lemma 1, $f(\bar{\tau}') = f(\bar{\tau}'')$.

Lemma 4. *Removing the first step of an incomplete Pop does not affect the result of f.*

Proof. The first step of an incomplete pop operation is never considered when building the set A, nor when selecting a pop operation out of A, so removing it will cause no change in the computation of f.

Lemma 5. *Let σ be the BH state at the end of a pop operation by some process P_d. Let σ' be a sequence indistinguishable to P_d from σ. Then $f(\sigma) = f(\sigma')$.*

Proof. By properties of the BH object, there is a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_m$ with $\sigma = \sigma_0$ and $\sigma_m = \sigma'$ such that any two consecutive states σ_a, σ_{a+1} can be obtained from one another by swapping two consecutive last steps by some processes other than P_d. We have three possibilities:

– One of these steps is a first pop step. Since it is the last step by that process, it must be part of an incomplete pop operation. By Lemma 4, removing it will not affect the result of f. But removing it erases the difference between σ_a and σ_{a+1}, so $f(\sigma_a) = f(\sigma_{a+1})$.
– Both steps are second pop steps. By Lemma 3, $f(\sigma_a) = f(\sigma_{a+1})$.
– One is a push step, the other is a second pop step. By Lemma 2, $f(\sigma_a) = f(\sigma_{a+1})$.

Inductively, $f(\sigma) = f(\sigma')$.

Next, we assign linearization points for Push operations and for completed Pop operations. We are not linearizing any incomplete Pop operations (which only apply one Sign). Push operations are linearized when their single Sign operation is performed. Let α be a complete Pop operation and let σ be the BH state when α is completed. By Lemma 5, we may assume that $f(\sigma)$ is computed as part of α. We define the linearization point of α as follows:

– If there are Push operations deleted unmatched (i.e. in line 19) during the computation of f on σ, let ϕ be the earliest such Push operation. We linearize α at the only step of ϕ, before ϕ itself. Multiple Pop operations linearized at the only step of ϕ are ordered by the locations of their second steps. Note that the only step of ϕ follows the first step of α, for otherwise α itself would be matched with ϕ.
– Otherwise, α is linearized at its second step.

Given σ, we define $h(\sigma)$ to be the Stack history associated with σ, containing the sequence of operations in the order they are linearized, together with their return values. For example, $h(11216\underline{2}\underline{6}42\underline{4}1\underline{2}6\underline{6})$ is the sequence (Push, OK), (Push, OK), (Pop (by P_2), 2), (Push, OK), (Pop (by P_6), 3), (Pop (by P_4), 1), (Pop (by P_2), ε), (Push, OK), (Pop (by P_6), 4).

Theorem 2. *For every state σ, the Stack history $h(\sigma)$ is legal.*

Proof. We use induction on the number of push steps in σ.

First, let σ be a history with no push steps. Any Pop operation which is completed during σ will output ε, hence $h(\sigma)$ is legal.

Now let $k \geq 0$ and assume that for all sequences σ' with at most k push steps, $h(\sigma')$ is legal. Let σ be a history with $k+1$ push steps. Let ϕ denote the last Push operation.

First, consider the case where σ contains no completed Pop operation that starts after the last push step. Let us write σ as $\tau_0, 1, \tau_1$, where τ_1 contains no push steps.

We claim that $h(\sigma)$ ends with ϕ. To see this, notice how the linearization points for Pop operations can be either at a second pop step, or at a push step. A Pop operation linearized at a push step occurs before the Push operation. Furthermore, a Pop operation cannot be linearized at a second pop step from τ_1, for it would have to have started after the last push step in σ, a situation ruled out by the case under consideration.

Now consider the history $\sigma' = \tau_0, \tau_1$. We claim that $h(\sigma) = h(\sigma'), (\text{Push}, \text{OK})$. The only operations whose linearization point could be different in σ and in σ' are Pop operations started in τ_0, finished in τ_1, and linearized in σ at the last push step. Notice, however, that their *order* is exactly the same in σ and in σ', namely the one determined by their second steps.

To establish our claim, we now have to argue that every Pop operation outputs the same result in σ and in σ'. The only non-trivial situation is when the second pop step of α occurs in τ_1. In the case under consideration, the first pop step of α must occur in τ_0. Then, during the first iteration of the `while` loop in f, the last push step is unmatched and deleted in line 19. This erases the only difference between the histories σ and σ', and hence, α outputs the same result in both. Therefore, $h(\sigma) = h(\sigma'), (\text{Push}, \text{OK})$.

By the induction hypothesis, $h(\sigma')$ is legal. Thus, so is $h(\sigma)$.

Now consider the case where σ contains at least one completed Pop operation that starts after the last push step. We write σ as $\tau_0, 1, \tau_1, d_1, \tau_2, d_2, \tau_3$, where d_1 and d_2 are the two steps of the first Pop operation α which starts after the last push step and is completed. Informally, we show that the linearization point of α immediately follows that of the last Push operation. We then argue that removing both of these operations will not cause any output values to change. By the induction hypothesis, the Stack history obtained after removing the last Push operation is legal. Adding two consecutive operations, a Push immediately followed by a matching Pop will preserve legality.

First we claim that the linearization point of α immediately follows that of ϕ. The reasons is that if some Pop operaion α' were linearized at a step in τ_1 or τ_2, that step would have to be the second step of α', and α' would have to have started after the last push step. This would contradict our choice of α.

Let $\sigma' = \tau_0, \tau_1, \tau_2, \tau_3$. We want to show that the order between any two operations other than ϕ and α is the same in σ and in σ'. We classify Pop operations as follows:

- Type I Pop operations are those with their second step in τ_0. Their linearization points are the same in σ and in σ' because they do not see the difference between the two.
- Type II Pop operations have their second step in either τ_1 or τ_2, and are linearized at the last push step in σ. By choice of α, any type II Pop operation must have its first step in τ_0.
- Type III Pop operations have their second step in either τ_1 or τ_2, but they are linearized at some earlier push step in σ. By choice of α, any type III Pop operation must have its first step in τ_0. Their linearization points are the same in σ and σ' because the same Push operations which are unmatched in σ will be unmatched in σ'.

– Type IV Pop operations have their second step in τ_3. Their linearization points are the same in σ and in σ', because ϕ and α are matched and deleted in the first iteration of f when called by any such Pop operation.

So the only operations whose linearization point might change are type II Pop operations. These are operations which are linearized at the last push step in σ in the order of their second steps; and at their second steps in σ'. Clearly, the ordering between any such two Pop operations remains the same. To see that the order between two Pop operations α' and α'', exactly one of which (say, α') is type II, remains unchanged, it is enough to point out that the linearization step of α' changes from the last push step to its second step in τ_1 or τ_2, while the linearization step of α'' is either in τ_0 (types I, III or IV) or in τ_3 (type IV).

Next, we claim that any Pop operation outputs the same result in σ and in σ'. For type I Pop operations, this is obvious, for they cannot tell the difference between σ and σ'.

For a type II or III Pop operation α', by Lemma 5, f is run on the appropriate prefixes $\sigma|_{\alpha'}$ and $\sigma'|_{\alpha'}$. But then, ϕ is unmatched and deleted in the first iteration of $f(\sigma|_{\alpha'})$. The only difference between $\sigma|_{\alpha'}$ without the last push step and $\sigma'|_{\alpha'}$ is an eventual first step of α, but this can be eliminated by Lemma 4. Hence, α' outputs the same value in σ and σ'.

For a type IV Pop operation α', again by Lemma 5, f is on the appropriate prefixes $\sigma|_{\alpha'}$ and $\sigma'|_{\alpha'}$. In the first iteration of $f(\sigma|_{\alpha'})$, ϕ and α are matched and deleted, erasing the difference between σ and σ'. Hence, α' outputs the same value in σ and σ'.

Last but not least, we note that α outputs the value pushed by ϕ. By Lemma 5, α runs f on the sequnce $\tau_0, 1, \tau_1, d_1, \tau_2, d_2$. By choice of α, ϕ will be matched with α in the first iteration of the `while` loop.

We have now established that $h(\sigma)$ is exactly equal to $h(\sigma')$ with an inserted pair of consecutive operations, the Push ϕ and the associated Pop α. By properties of a Stack object, if $h(\sigma')$ is legal, then $h(\sigma)$ is legal.

3.2 A single-valued Stack implementation

The single-popper Stack implementation is based on the observation that the number of times each pusher signs the BH object prior to a Pop is precisely the number of elements that were pushed on the Stack prior to that Pop. If there is only one pusher, there is no ambiguity about the order in which the Push operations occurred. Unfortunately, this is not the case when there are many pushers. For example, suppose processes P_1 and P_2 each pushed a value on the Stack by signing the BH object and then process P_3 popped a value by signing the BH object twice. The resulting state 1233 of the BH object is indistinguishable to P_3 from 2133, the state that results when P_1 and P_2 perform their operations in the opposite order. Consequently, it is not clear if the value pushed by P_1 or P_2 is the one which should be popped. While we can overcome this problem for the special case of exactly two pushers (see following section), the general solution remains elusive. However, if all the values pushed on the Stack are the same, then the problem of choosing which value to match with which Pop is obviated.

A process performing a Pop on a single-valued Stack only needs to determine whether or not its Pop operation has some matching Push. It does not matter which pusher performed the Push. This is essentially the problem that is solved by the single-pusher Stack implementation (in the previous section).

To perform a Push, a process appends 1 to its single-writer Register and Signs the BH object once. To perform a Pop, a process appends 22 to its single-writer Register, Signs the BH object twice, and then reads the Registers of all other processes. Let C denote the equivalence class of BH states returned as a result of the second Sign operation in a Pop. As in line 6, we select any representative σ from C. However, before we compute f on σ, we replace every push step in σ with a push step by a virtual process, P_0. A certain step by some process P_a is a push step if the corresponding value in P_a's Register is a 1. If f on the modified sequence returns 0, the Pop returns ε; otherwise the Pop returns the single value in the domain.

The proof of correctness is, with a minor exception, identical to the proof for the single-pusher Stack. We need one more lemma that deals with two Push operations whose order can not be distinguished.

Lemma 6. *Swapping two consecutive push steps does not affect the result of f.*

Proof. Since the selection of a push step is independent of its label, a swap of two consecutive push steps simply corresponds to a relabeling of the Push operations. Consequently, the only thing that changes is the label of a Push that is discarded or matched with a Pop. Hence, inductively, the two computations, one on the original sequence and one on the sequence with the swap, take exactly the same decision during every iteration of the `while` loop.

3.3 A two-pusher Stack implementation

We will now extend the algorithm from the previous section to allow two pushers instead of just one. Informallly, the basic idea is similar to the "helping" mechanism that appears in Herlihy's universal construction [Her91]: the completion of a Push operation by one pusher might "help" linearize a pending Push operation by the other pusher.

Let P_1 and P_2 be the two pushers, and let P_d be the poppers, for $d > 2$. We assume that in addition to a BH object B, we have two unbounded arrays V_1, V_2 of SW Registers, with V_a written by pusher P_a. To push the value x, P_a first writes x in the next avaliable location inf V_a. P_a then applies *two* Sign operations on B. Recall that in the single-pusher implementation, a Push operation consisted of only one Sign.

A Pop operation by P_d begins by applying two Sign operations on the BH object. The return value of the second operation is an equivalence class of states indistinguishable to P_d from the real state of B. We then select any representative σ, as in line 6. However, before we can apply function f on σ, we need to transform σ from a two-pusher, 2-step/Push history into a single-pusher, 1-step/Push history. This transformation is performed by a new function, g, described below.

The function g takes as arguments a 2-pusher 2-step/Push history σ, and the arrays V_1, V_2. It constructs a single-pusher 1-step/Push history τ and an array V. The two histories, σ and τ, contain exactly the same pop steps. The push steps by P_1 and P_2 in σ are replaced in τ with push steps by a virtual process, P_0. The idea is that a Push operation ϕ is linearized either at its second step, or at the second step of the first push operation ϕ' by the other pusher which was started and completed after the first step of ϕ. The function g performs the following:

- Find the earliest second push step in σ, call that push operation ϕ';
- If there is a push operation ϕ which has a first step before the first push step of ϕ', delete both ϕ and ϕ', and insert two steps by P_0 in τ at the location of the second push step of ϕ'.
- If no such ϕ exists, delete ϕ' and insert a step by P_0 in τ at the location of the second push step of ϕ'.
- Whenever we delete the i-th Push operation by P_a, append $V_a[i]$ to V. In the first case, when we delete ϕ and ϕ', append the value corresponding to ϕ before the one corresponding to ϕ'.
- Repeat until no push operation in σ has two steps.
- At the end, delete the remaining first push steps.

For the purposes of proving correctness, we may assume that the i-th value pushed by P_a, and written in $V_a[i]$, is the pair (a,i). For example, if $\sigma = 1\underline{1}123\underline{3}\underline{2}6\underline{1}124\underline{1}3\underline{3}\underline{2}2\underline{4}314\underline{2}\underline{6}$ (where second push and pop steps are underlined), we have $g(\sigma,V_1,V_2) = (\tau,V)$ where $\tau = 03\underline{3}0064033\underline{3}0\underline{4}240\underline{6}$ and V = (1,1), (1,2), (2,1), (1,3), (2,2), (2,3).

After computing $g(\sigma) = (\tau,V)$, a Pop operation computes $f(\tau)$. If the latter evaluates to 0, the Pop returns ε; otherwise the Pop returns the element at location $f(\tau)$ from V, the array computed in g. For example, for σ,τ,V from the previous example, $f(\tau) = 2$ and $V[f(\tau)] = (1,2)$.

The following two Lemmas are needed to prove the correctness of this extension.

Lemma 7. *Let σ be the state at the end of a Pop operation by P_d, let σ' be a state indistinguishable to P_d from σ. Let $g(\sigma) = (\tau,V)$ and $g(\sigma') = (\tau',V')$. Then $V = V'$ and $f(\tau) = f(\tau')$.*

Lemma 8. *Let σ be a BH state. Let σ' be any prefix of σ. Let $g(\sigma) = (\tau,V)$ and let $g(\sigma') = (\tau',V')$. Then τ' is a prefix of τ and V' is a prefix of V.*

Finally, we argue that our algorithm is linearizable. Given a two-pusher 2-step/Push history σ, let $g(\sigma) = (\tau,V)$. We define the linearization points for Push operations in σ to be the corresponding steps where they appear in τ. We define linearization points for Pop operations in σ the same way they are defined in the single-pusher history τ. By Lemma 8, all Pop operations completed in σ have output the exact same values as if they had occurred in τ. Since the single-pusher 1-step/Push history τ is linearizable, so is σ.

4 Conclusions

In this paper, we have showed that several restricted Stack and Queue implementations exist from Registers and Common2 objects. Specifically, it is possible to implement single-valued Stacks and Queues shared by any number of process, and general (multi-valued) Stacks shared by one or two pushers and any number of poppers.

Queue implementations exist for any number of enqueuers and at most two dequeuers [Li01], and for one enqueuer and any number of dequeuers [Dav04a]. In a Stack implementation, only the poppers output relevant values. Informally, if there are only two poppers, they would probably be able to agree on the sequence of values to output. This suggests that Stack implamentations for any number of pushers and at most two poppers might exist. However, we believe that implementing a Stack shared by three pushers, three poppers, with domain size 2 is impossible to implement from Registers and Common2 objects.

We also introduce the BH object type. Although unusable in practice, this type encapsulates the computational power of a system with Registers and Common2 objects, and has the potential of helping in the development of negative results regarding wait-free implementations in this system.

References

[AWW93] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 159–170, 1993.

[Dav04a] Matei David. A single-enqueuer wait-free queue implementation. In *Proceedings of DISC 2004*, pages 132–143, 2004.

[Dav04b] Matei David. Wait-free linearizable queue implementations. Master's thesis, Univ. of Toronto, 2004.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HW90] Maurice Herlihy and Jeanette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):495–504, January 1990.

[Li01] Zongpeng Li. Non-blocking implementation of queues in asynchronous distributed shared-memory systems. Master's thesis, Univ. of Toronto, 2001.