Universal Constructions for Large Objects*

James H. Anderson
Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599
anderson@cs.unc.edu

Mark Moir
Department of Computer Science
The University of Pittsburgh
Pittsburgh, PA 15260
moir@cs.pitt.edu

June 1997; revised March 1999; accepted June 1999

Abstract

We present lock-free and wait-free universal constructions for implementing large shared objects. Most previous universal constructions require processes to copy the entire object state, which is impractical for large objects. Previous attempts to address this problem require programmers to explicitly fragment large objects into smaller, more manageable pieces, paying particular attention to how such pieces are copied. In contrast, our constructions are designed to largely shield programmers from this fragmentation. Furthermore, for many objects, our constructions result in lower copying overhead than previous ones.

Fragmentation is achieved in our constructions through the use of *load-linked*, *store-conditional*, and *validate* operations on a "large" multi-word shared variable. Before presenting our constructions, we show how these operations can be efficiently implemented from similar one-word primitives.

Keywords: concurrency, lock-free, non-blocking synchronization, shared objects, wait-free.

*Work supported by NSF grants CCR 9216421, CCR 9510156, and CCR 9732916, and by a Young Investigator Award from the U.S. Army Research Office, grant number DAAH04-95-1-0323. The first author was also supported by an Alfred P. Sloan research fellowship. The second author was also supported by a UNC Alumni Fellowship, by an NSF CAREER Award, CCR 9702767, and by an ORAU Junior Faculty Enhancement Award. A preliminary version of this work appeared in the *Proceedings of the Ninth International Workshop on Distributed Algorithms*, September 1995.

1 Introduction

This paper extends recent research on *universal* lock-free and wait-free constructions of shared objects [7, 8]. An object implementation is *wait-free* if *every* operation by each process is guaranteed to complete after a finite number of steps of that process. An object implementation is *lock-free* if *some* operation is guaranteed to complete after a finite number of steps of any operation. Universal constructions can be used to implement any object in a lock-free or a wait-free manner, and thus can be used as the basis for a general methodology for constructing highly-concurrent objects. Unfortunately, the generality of universal constructions often comes at a price, specifically space and time overhead that is excessive for many objects. A particular source of inefficiency in previous universal constructions is that they require processes to copy the entire object state, which is impractical for large objects. In this paper, we address this shortcoming by presenting universal constructions that can be used to implement large objects with low space overhead.

We take as our starting point the lock-free and wait-free universal constructions presented by Herlihy in [8]. In these constructions, operations are implemented using "retry loops". In Herlihy's lock-free universal construction, each process's retry loop consists of the following steps: first, a shared object pointer is read using a *load-linked* (LL) operation, and a private copy of the object is made; then, the desired operation is performed on the private copy; finally, a *store-conditional* (SC) operation is executed to attempt to modify the shared object pointer so that it points to the private copy. (See Section 2 for formal definitions of LL and SC.) The SC operation may fail, in which case these steps are repeated. This algorithm is not wait-free because the SC of each loop iteration may fail. To ensure termination, Herlihy's wait-free construction employs a "helping" mechanism, whereby each process attempts to help other processes by performing their pending operations together with its own. This mechanism ensures that if a process is repeatedly unsuccessful in modifying the shared object pointer, then it is eventually helped by another process (in fact, after at most two loop iterations).

As Herlihy points out, these constructions perform poorly if used to implement large objects. To overcome this problem, he discusses techniques for implementing large objects. The basic idea is to fragment a large object into blocks linked by pointers. Operations are implemented so that blocks are copied only if they are modified. Herlihy illustrated these large-object techniques by presenting an implementation of an example object, a skew heap.

Herlihy's lock-free approach for implementing large objects suffers from three shortcomings. First, the required fragmentation is left to the programmer to determine, based on the semantics of the implemented object. The programmer must also explicitly determine how copying is done. Second, Herlihy's large-object approach is difficult to apply in wait-free implementations. In particular, directly combining it with the helping mechanism of his wait-free construction results in excessive space overhead. Third, Herlihy's large-object approach reduces copying overhead only if long chains of linked blocks are avoided. Consider, for example, a large shared queue that is fragmented as a linear sequence of blocks (i.e., in a linked list). Replacing the last block actually requires the replacement of every block in the sequence. In particular, linking in a new last block requires that the pointer in the previous block be changed. Thus, the next-to-last block must be replaced. Repeating this argument, it follows that every block must be replaced.

Our approach for implementing large objects is also based upon the idea of fragmenting an object into blocks. However, it differs from Herlihy's in that it is array-based rather than pointer-based, i.e., we view a large object as a long array that is fragmented into blocks. In its retry loop, a process accesses only the blocks that must be read or written to perform the operation(s) it is attempting to apply. Each block that contains an array location that must be written is copied to a private block, and the *copy* is written, rather than the block that is still part of the array. These writes are made to take effect globally by atomically updating a collection of shared block pointers so that the newly-written blocks become part of the array. Unlike Herlihy's approach, the fragmentation in our approach is determined by our constructions, and is not visible to the user. Also, copying overhead in our approach is often much lower than in Herlihy's approach.

Our constructions are similar to Herlihy's in that operations are performed using retry loops. However, while Herlihy's constructions employ only a single shared object pointer, we need to manage a collection of such pointers, one for each block of the array. We deal with this problem by employing LL, SC, and *validate* (VL) operations that access a large shared variable that contains all of the block pointers. This large variable is stored across several memory words.[1] In the first part of the paper, we show how to efficiently implement these operations using the usual single-word LL, SC, and VL primitives. We present an implementation in which LL and SC operations on a W-word variable take $O(W)$ time and VL takes constant time. Our implementation allows LL to return a special value that indicates that a subsequent SC will fail — we call this a *weak*-LL. This relaxed requirement admits simpler and more efficient implementations because weak-LL does not have to return a consistent multi-word value in the case of interference by a concurrent SC. Also, weak-LL can be used to avoid unnecessary work in universal algorithms (there is no point in completing a retry-loop iteration if the SC of that loop iteration is certain to fail). For these reasons, we use weak-LL in our universal constructions. (Elsewhere, we have presented a similar implementation that provides the "normal" semantics for the LL operation [3]. That implementation has the same asymptotic time complexity as the one presented here, but is more complicated and therefore likely to perform worse. It also requires $O(N^2W)$ space to implement a W-word variable for N processes, while the one presented here requires only $O(NW)$ space.)

To avoid the excessive space requirements of previous wait-free universal constructions, our wait-free construction imposes an upper bound on the number of private blocks each process may have. This bound is assumed to be large enough to accommodate any two operations. The bound affects the manner in which processes may help one another. Specifically, if a process attempts to help too many other processes simultaneously, then it runs the risk of using more private space than is available. We solve this problem by having each process help as many processes as possible with each operation, and by choosing processes to help in such a way that all processes are eventually helped. If enough space is available, all processes can be helped by one process at the same time — we call this *parallel* helping. Otherwise, several "rounds" of helping must be performed, possibly by several processes — we call this *serial* helping. The tradeoff between serial and parallel helping is one of time versus space.

We are aware of only two other wait-free universal constructions that do not copy the entire object for each operation. The first one is due to Afek et al. [1], and the second is due to Moir [11]. The main feature of the construction of Afek et al. is that its worst-case time complexity is dependent on the level of contention (i.e., the number of processes that access the object simultaneously), rather than the total number of processes in the system. The main feature of Moir's construction is that it allows operations to execute in parallel where possible. While these constructions represent important progress, they both make heavy use of the LL and SC synchronization operations and are quite complicated. The construction of [1] has not been implemented. As discussed in Section 7, simulation results presented in [5] suggest that the complexity of the construction in [11] causes it to perform worse than ours at least for some objects.

The remainder of this paper is organized as follows. In Section 2, we outline our model of computation and the correctness conditions for our results. In Section 3, we present implementations of the weak-LL, SC, and VL operations for large variables discussed above. We then present our lock-free and wait-free universal constructions in Sections 4 and 5, respectively. Finally, we discuss performance results in Section 6, and end the paper with concluding remarks in Section 7. Formal correctness proofs have been published elsewhere [10], and are long and tedious, so we do not present them again here. However, we do give a high-level, intuitive proof sketch in an appendix.

[1]The multi-word operations considered here access a *single* variable that spans multiple words. The multi-word operations considered in [2, 4, 9, 13] are more general: they can access *multiple* variables, each stored in a separate word. These operations could therefore be used to implement the operations we require. However, we implement the single-variable, multi-word operations used by our construction directly because they admit simpler and more efficient implementations than those considered in [2, 4, 9, 13].

2 Preliminaries

Our algorithms are designed for use in shared-memory multiprocessors that provide load-linked (LL), validate (VL), and store-conditional (SC) instructions. We assume that the SC operation does not fail spuriously. (In some hardware implementations if LL and SC, SC can fail even when the semantics of LL and SC — given below — dictates that it should succeed. We call such failures *spurious failures.*) As shown in [2, 11], algorithms based on these assumptions are applicable in machines that provide either compare-and-swap, or a limited form of LL and SC that is commonly available in hardware. The semantics of LL, VL, and SC are shown by equivalent atomic code fragments below.

```
LL(X) ≡ valid_X[p] := true;            SC(X, v) ≡ if valid_X[p] then
        return X                                       X := v;
                                                       for i := 0 to N − 1 do valid_X[i] := false od;
                                                       return true
                                                   else return false
VL(X) ≡ return valid_X[p]                          fi
```

These code fragments are for process p. $valid_X$ is a shared array of booleans associated with variable X. i is a private variable of process p. N is the total number of processes. The semantics of VL and SC are undefined if process p has not executed a LL instruction since p's most recent SC.

3 LL and SC on Large Variables

In this section, we implement weak-LL, VL, and SC operations for a W-word variable V, where $W > 1$, using the standard, one-word LL, VL, and SC operations. Recall that weak-LL can return a failure value — instead of a correct value of the implemented variable — in the case that a subsequent SC operation will fail. Nonetheless, weak-LL is suitable for many applications. In particular, in most lock-free and wait-free universal constructions (including the ones presented in Sections 4 and 5), LL and SC are used in pairs in such a way that if a SC fails, then none of the computation since the preceding LL has any effect on the object. By using weak-LL, we can avoid such unnecessary computation.

In the implementation presented in this section, if a subsequent SC is guaranteed to fail, then weak-LL returns the process identifier of some process that performed a successful SC during the execution of the weak-LL operation. We call this process a *witness* of the failed weak-LL. As we will see in Section 5, the witness of a failed weak-LL provides useful information that held during the execution of that weak-LL.

Our implementation of weak-LL, VL, and SC for large variables is shown in Figure 1.[2] The *Long_Weak_LL* and *Long_SC* procedures implement weak-LL and SC operations on a W-word variable V. Values of V are stored in buffers, and a shared variable X indicates which buffer contains the current value of V. The *current value* of V is the value written to V by the most recent successful SC operation, or the initial value of V if there is no preceding successful SC. We call the buffer that contains the current value of V the *current buffer*. The VL operation for V is implemented by simply validating X.

A SC operation on V is achieved by writing the W-word variable to be stored into a buffer, and then using a one-word SC operation on X to make that buffer current. To ensure that a SC operation does not overwrite the contents of the current buffer, the SC operations of each process p alternate between two buffers, $BUF[p, 0]$ and $BUF[p, 1]$. To see why this ensures that the current buffer is not overwritten, observe that, if $BUF[p, 0]$ is the current buffer, then process p will attempt a SC operation using $BUF[p, 1]$ before it modifies $BUF[p, 0]$ again. If p's SC succeeds, then $BUF[p, 0]$ is no longer the current buffer. If p's SC fails, then some other process has performed a successful SC, which also implies that $BUF[p, 0]$ is no longer the current buffer.

[2] Private variables in all figures are assumed to retain their values between procedure calls.

```
shared variable X: record pid: 0..N − 1; tag: 0..1 end;
                BUF: array[0..N − 1, 0..1] of array[0..W − 1] of wordtype
initially X = (0,0) ∧ BUF[0,0] = initial value of the implemented variable V

private variable curr: record pid: 0..N − 1; tag: 0..1 end; i: 0..W − 1; side: 0..1
initially side = 0

procedure Long_Weak_LL(var r: array[0..W − 1]
                        of wordtype) returns 0..N
1:  curr := LL(X);
    for i := 0 to W − 1 do
2:      r[i] := BUF[curr.pid, curr.tag][i]
    od;
3:  if VL(X) then return N
4:  else return X.pid
    fi

procedure Long_SC(val: array[0..W − 1] of wordtype)
                        returns boolean
4:  side := 1 − side;
    for i := 0 to W − 1 do
5:      BUF[p, side][i] := val[i]
    od;
6:  return SC(X, (p, side))
```

Figure 1: W-word weak-LL and SC using 1-word LL, VL, and SC. W-word VL is implemented by validating X.

A process p performs a weak-LL operation on V in three steps: first, it executes a one-word LL operation on X to determine which buffer contains the current value of V; second, it reads the contents of that buffer; and third, it performs a VL on X to check whether that buffer is still current. If the VL succeeds, then the buffer was not modified during p's read, and the value read by p from that buffer can be safely returned. (Note that this value is returned by means of the **var** parameter r. This avoids redundant copying of the returned values.) In this case, weak-LL returns N (which is *not* a valid witness process identifier) to indicate that it has obtained a consistent value of the implemented variable. If the VL fails, then the weak-LL re-reads X at line 4 in order to determine the identity of the last process to perform a successful SC; this process identifier is then returned. Note that if the VL of line 3 fails, then a subsequent SC by p will fail. The implementations of the weak-LL, VL, and SC operations for a W-word variable shown in Figure 1 have time complexity $\Theta(W)$, $\Theta(1)$, and $\Theta(W)$, respectively, and space complexity $\Theta(NW)$.

4 Lock-Free Universal Construction for Large Objects

The lock-free construction presented in this section provides the object programmer with a general framework for the implementation of lock-free shared objects. To use this construction, a programmer writes for each required operation a sequential procedure that treats the object as if it were stored in a contiguous array. In order to invoke an operation, the programmer then passes a pointer to the procedure to our construction (by means of the *LF_Op* procedure presented later). Our construction calls the programmer-supplied procedure and intercepts each access to the array in order to maintain the data structures required to make the operation lock-free. Unlike Herlihy's small-object constructions, the array that stores object values is not actually stored in contiguous locations of shared memory. Instead, we provide the *illusion* of a contiguous array, which is in fact partitioned into blocks. Our construction replaces only the blocks modified by the invoked operation, and thus avoids copying the whole object. Before discussing our lock-free construction in detail, we explain below how we provide the illusion of a contiguous array without copying the whole array.

Figure 2 shows an array MEM, which is divided into B blocks of S words each. Memory words $MEM[0]$ to $MEM[S-1]$ are stored in the first block, words $MEM[S]$ to $MEM[2S-1]$ are stored in the second block, and so on. A "bank" of pointers (called $BANK$), one to each block of the array, is maintained in order to record which blocks are currently part of the array. (As will be seen later, these "pointers" are really array indices.) While performing an operation, a process p maintains a logical view of the current array. This logical view is represented by a private array of pointers $ptrs$. At the beginning of the operation, $ptrs$ contains the same pointers that the $BANK$ array contains. However, if p's operation changes the contents of the array, then p makes a copy of each block to be changed, installs the copies into its logical view, and then

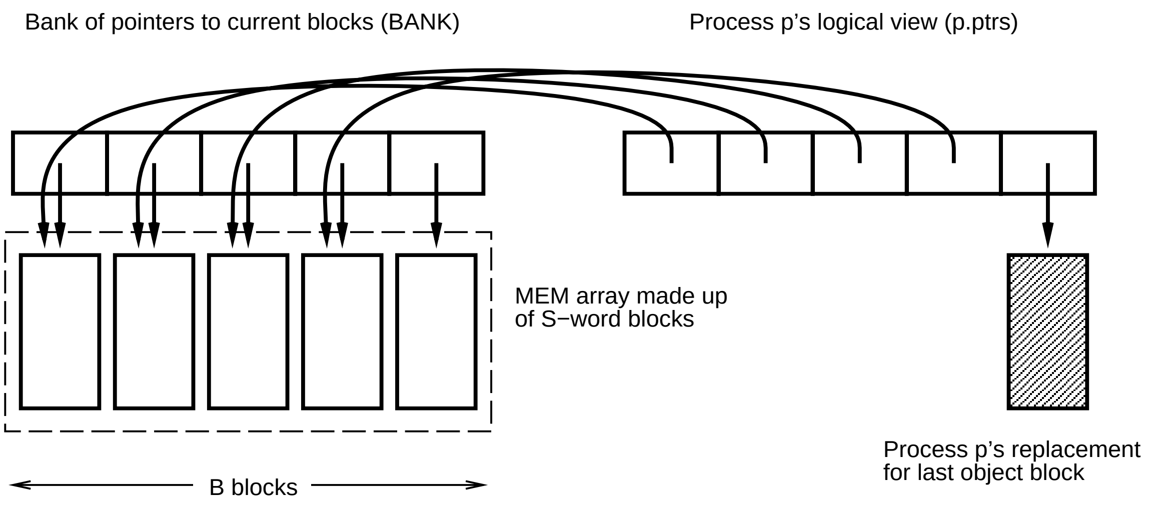

Bank of pointers to current blocks (BANK)
Process p's logical view (p.ptrs)
MEM array made up of S−word blocks
Process p's replacement for last object block
B blocks

Figure 2: Implementation of the *MEM* array for large-object constructions.

modifies the copied blocks, rather than the blocks that are part of the current array. Having completed its operation, process p then attempts to make its logical view of the array become the current view by writing the values in *ptrs* to *BANK*. (The *BANK* array is implemented through the use of the weak-LL and SC operations for large variables presented in Section 3. Thus, *BANK* is not a real variable, but rather a name for the set of values in the current buffer of the weak-LL and SC implementation. We present the algorithm this way because we think it aids intuition to separate the concerns of the weak-LL and SC implementation, and the algorithms that use it.) In Figure 2, process p's operation modifies the last block, but no others. Thus, the bank of pointers to be written by p is the same as the current bank, except that the last pointer points to p's new last block.

When an operation by process p accesses a word in the array, say $MEM[k]$, the block that currently contains $MEM[k]$ must be identified. If p's operation modifies $MEM[k]$, then p must replace that block. In order to hide the details of identifying blocks and of replacing modified blocks, some address translation and record-keeping is necessary. This work is performed by special *Read* and *Write* procedures, which are called by the sequential operation in order to read or write the *MEM* array. Because sequential object code must call these procedures, our constructions are not completely transparent to the sequential object designer. For example, instead of writing "$MEM[1] := MEM[10]$", the designer would write "$Write(1, Read(10))$".

As a more concrete example of how a programmer would write the sequential code for an operation, consider Figure 3, which contains the actual code we used to implement a FIFO queue using our constructions. As seen in the figure, this code is very similar to the "normal" sequential code for a queue. Indeed, the differences between the sequential code and the code used with our constructions are syntactic in nature, so it should be easy to develop a preprocessor or compiler that automatically generates code for use with our constructions from sequential code. This would make the use of our constructions entirely transparent.

The code for our lock-free construction is shown in Figure 4.[3] Before explaining this code in detail, we first describe the data structures used, and present a simple example that illustrates how each process maintains a logical view of the object during the execution of operations.

As mentioned above, we assume that the object to be implemented fits into an array that is made up of B blocks of S words each. We also assume an upper bound T on the number of blocks modified by each operation. Each process has T "copy blocks", which it uses to maintain its logical view of the object during

[3] An extra parameter (the first one) has been added to the procedures of Section 3 to explicitly indicate that these procedures are being used to update the bank of pointers. *memcpy* is a standard C library call, which copies the values from one area of memory to another. This copying is not necessarily executed atomically.

```
int enqueue(item)
     int item;
{
 int newtail;                    /* int newtail;           */

 Write(Read(tail),item);         /* MEM[tail] = item;      */
 newtail = (Read(tail)+1)%n;     /* newtail = (tail+1) % n; */
 if (newtail == Read(head))      /* if (newtail == head)   */
   return FULL;                  /*   return FULL;         */
 Write(tail,newtail);            /* tail = newtail;        */
 return SUCCESS;                 /* return SUCCESS;        */
}
```

Figure 3: C code used for the enqueue operation of an array-based queue implementation. n is the maximum size of the queue. **head** and **tail** are constants that are defined to be **n** and **n+1**, respectively. Thus, these variables denote array locations outside the area used for data elements. Comments show "normal" enqueue code.

```
constant
    $N$ = number of processes
    $B$ = number of blocks in shared object
    $S$ = block size in words
    $T$ = maximum number of blocks modified by an operation

type
    blktype = array[0..$S-1$] of wordtype

shared variable
    BANK: array[0..$B-1$] of 0..$B+N*T-1$;          /* Bank of pointers to array blocks */
    BLK: array[0..$B+N*T-1$] of blktype             /* Array and copy blocks */

initially ($\forall n: 0 \le n < B :: BANK[n] = N*T+n \wedge BLK[N*T+n]$ = ($n$th block of initial value))

private variable
    copy, oldlst: array[0..$T-1$] of 0..$B+N*T-1$; ptrs: array[0..$B-1$] of 0..$B+N*T-1$; dirty: array[0..$B-1$] of
    boolean; dcnt: 0..$T$; i, blkidx: 0..$B-1$; v: wordtype; ret: objrettype

initially ($\forall n: 0 \le n < T :: copy[n] = p*T+n$)

procedure Read(addr: 0..$B*S-1$) returns wordtype
1:  v := BLK[ptrs[addr div S]][addr mod S];
2:  if ¬VL(BANK) then goto 11 else return v fi

procedure Write(addr: 0..$B*S-1$; val: wordtype)
3:  blkidx := addr div S;
4:  if ¬dirty[blkidx] then
5:      memcpy(BLK[copy[dcnt]], BLK[ptrs[blkidx]],
                                        sizeof(blktype));
6:      dirty[blkidx] := true;
7:      oldlst[dcnt] := ptrs.blks[blkidx];
8:      ptrs.blks[blkidx] := copy[dcnt];
9:      dcnt := dcnt + 1
    fi;
10: BLK[ptrs[blkidx]][addr mod S] := val

procedure LF_Op(op: optype; pars: paramtype)
    while true do
11:     if Long_Weak_LL(BANK, ptrs) = N then
12:         for i := 0 to $B-1$ do dirty[i] := false od;
13:         dcnt := 0;
14:         ret := op(pars);
15:         if dcnt = 0 ∧ Long_VL(BANK) then return ret fi;
16:         if Long_SC(BANK, ptrs) then
17:             for i := 0 to dcnt − 1 do copy[i] := oldlst[i] od;
18:             return ret
            fi
        fi
    od
```

Figure 4: Lock-free implementation for a large object.

an operation. Thus, a total of $B + NT$ blocks are required. These blocks are stored in the BLK array. $BLK[NT]$ to $BLK[NT + B - 1]$ are the initial array blocks, and $BLK[pT]$ to $BLK[(p+1)T - 1]$ are process p's initial copy blocks. The blocks that make up the current value of the array are recorded in the B-word shared variable $BANK$, and the copy blocks for each process p are stored in p's private *copy* array.

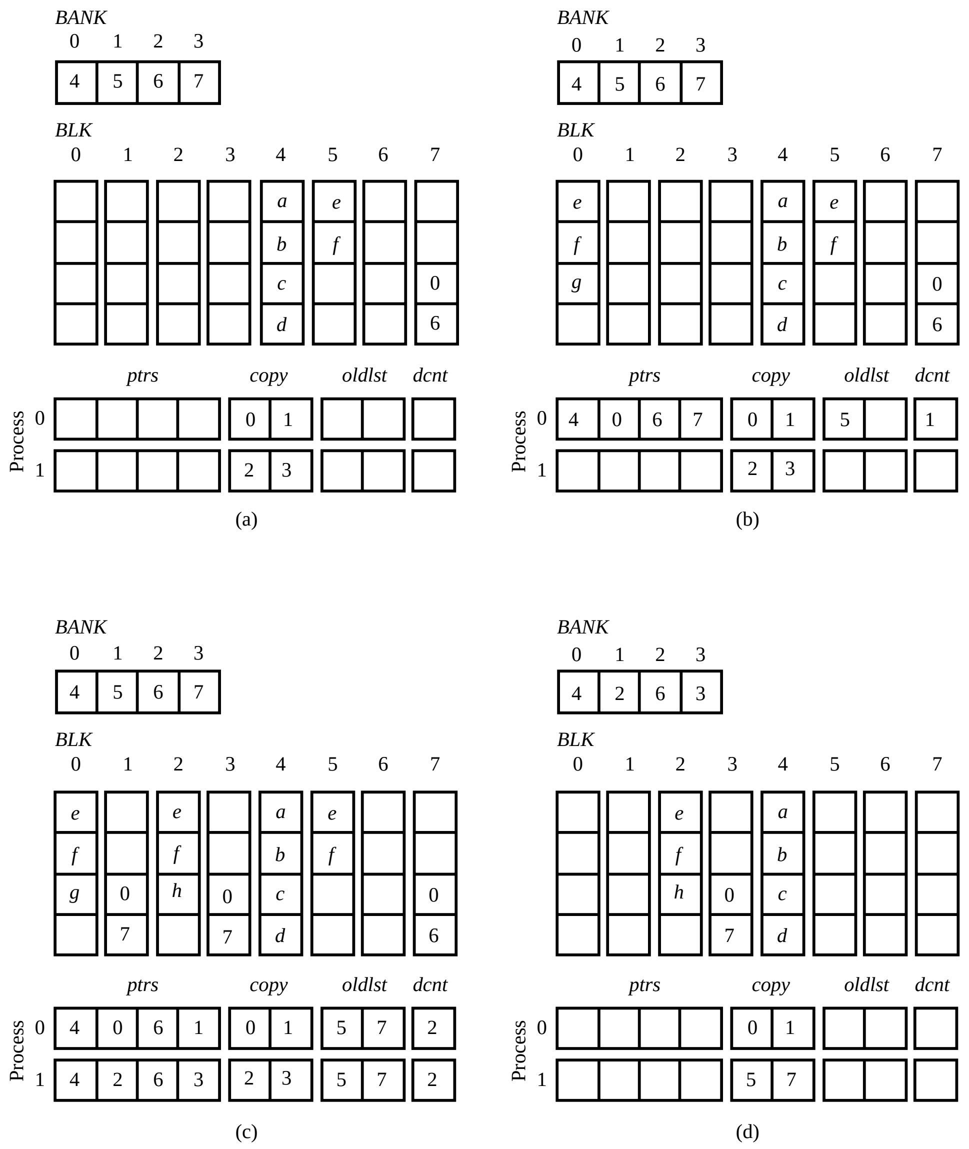


Figure 5: Example execution showing two concurrent enqueue operations on a FIFO queue.

Figures 5(a) through 5(d) show several states of the variables used by our construction when implementing a lock-free FIFO queue for two processes. In this example, the queue is represented by four blocks of four locations each. The last two locations are used for head and tail pointers. These locations denote the position of the first element in the queue, and the position of the next element to be enqueued, respectively. The queue initially contains $\{a, b, c, d, e, f\}$. Figure 5(a) shows the initial state. Blocks 4 through 7 of BLK are the initial blocks representing the queue, $BANK$ points to these blocks, and the *copy* arrays record that the initial copy blocks of process 0 are blocks 0 and 1, and the initial copy blocks of process 1 are blocks 2 and 3. However, the roles of these blocks are not fixed. In particular, if an operation by process p replaces a set of array blocks with some of its copy blocks, then, as explained below, p reclaims the replaced array blocks as copy blocks. Thus, the copy blocks of one process may become blocks of the array, and later become copy blocks of another process. This is illustrated by the example execution shown in Figures 5(b) through 5(d).

In Figure 5(b), process 0 begins an $enqueue(g)$ operation. It first reads the $BANK$ array into its private $ptrs$ array; this has the effect of setting process 0's logical view to be the current object value. It then begins executing its enqueue operation. As shown in Figure 3, the enqueue operation first reads the `tail` location and then inserts the new item g into the location pointed to by `tail`. As described below, our construction does the address-translation necessary to allow the read of `tail` to access the correct block. Also, when the enqueue operation writes g into logical array location 6, our construction intervenes and makes a copy of the affected block (block 5) in one of process 0's copy blocks (block 0), and modifies that block instead of block 5, which is still part of the current value of the object (as recorded by *BANK*). Our construction then records in process 0's *dcnt* variable that a block has been copied (*dcnt* counts blocks that have become "dirty" due to being written), records which block has been replaced in process 0's *oldlst* array, and updates process 0's logical view of the object by changing the second position of process 0's *ptrs* array. This results in the state shown in Figure 5(b). Now process 0's view of the object shows that g has been inserted into the second array block. However, this change has not yet been reflected in the current value of the object, which still has block 5 as its second block.

In Figure 5(c), process 0 has completed its sequential operation by updating the tail variable (stored in the last location of the logical array). As a result, a copy of the last array block has been made, and process 0's logical view has been updated as before. Also, process 1 has begun an *enqueue*(h) operation, and has made similar changes as process 0. Now, each process's logical view reflects the operation of that process, but *BANK* still records that blocks 4 through 7 are the current array blocks, so neither operation has taken effect yet.

Next, process 1 successfully writes the values in its *ptrs* array into *BANK*, thereby making its operation take effect. (See Figure 5(d).) Specifically, the *BANK* array now indicates that the second and fourth blocks of the current array are blocks 2 and 3 respectively; thus process 1's enqueue of h has taken effect. Furthermore, process 1 has "reclaimed" the blocks it displaced (blocks 5 and 7) as its new copy blocks. Because the *BANK* variable is modified by means of a *Long_SC* operation, when process 0 attempts to make its operation take effect (thereby potentially corrupting the object), it will fail to do so on account of process 0's successful SC. Process 0 will then start the operation again from the beginning. Having given an overview of our lock-free construction, we now describe its code, shown in Figure 4, in detail.

Process p performs a lock-free operation by calling the *LF_Op* procedure. The loop in the *LF_Op* procedure repeats until the SC at statement 16 succeeds. (Actually, read-only operations return from statement 15 because they do not need to modify any pointers, and they do not need to reclaim any copy blocks. This optimization allows read-only operations to execute in parallel with other operations because they do not perform the SC operation at statement 16, and therefore do not cause the SC operations of other processes to fail.) In each iteration, process p first reads *BANK* into p's private variable *ptrs* using a B-word weak-LL (statement 11). Recall from Section 3 that the weak-LL can return a process identifier from $\{0, ..., N-1\}$ if the following SC is guaranteed to fail. In this case, there is no point in attempting to apply p's operation, so the loop is restarted. Otherwise, p records in its *dirty* array that no block has yet been modified by its operation (statement 12), and initializes its *dcnt* counter to zero (statement 13). Process p uses its *dirty* array and its *dcnt* counter to record which blocks of its logical view are not part of the current array.

Next, p calls the *op* procedure provided as a parameter to *LF_Op* (statement 14). The *op* procedure performs the sequential operation by reading and writing the elements of the *BLK* array. (The programmer thinks of the operation as if it were modifying the implemented *MEM* array; as explained below, the construction determines which of the blocks in *BLK* currently represents the block of *MEM* being accessed, and modifies that block.) This reading and writing is performed by invoking the *Read* and *Write* procedures shown in Figure 4. The *Read* procedure simply computes which block currently contains the word to be accessed, and returns the value from the appropriate offset within that block. In order to perform a write to a word of *MEM*, the *Write* procedure first computes the index *blkidx* of the block containing the word to be written (statement 3). Then, if it has not already done so (statement 4), the *Write* procedure copies the contents of the old block to one of p's copy blocks (statement 5) and records that the block is "dirty" — i.e., has been modified (statement 6). Then, the displaced block is recorded in *oldlst* for possible reclaiming later (statement 7), the copy block is linked into p's *ptrs*, making that block part of p's logical view of the *MEM* array (statement 8), and p's *dcnt* variable is incremented in order to count the number of blocks replaces (statement 9). Finally, the appropriate word of the new block is modified to contain the value passed to the *Write* procedure (statement 10).

If *BANK* is not modified by another process after p's weak-LL, then the object contained in p's version of the *MEM* array (pointed to by p's *ptrs* array) is the correct result of applying p's operation. Therefore, p's SC successfully installs a copy of the object with p's operation applied to it. After the SC, p reclaims the displaced blocks (recorded in *oldlst*) to replace the copy blocks it used in performing its operation. On the other hand, if another process *does* modify *BANK* between p's weak-LL and SC, then p's SC fails. In this case, some other process completes an operation. Therefore, the implementation is lock-free.

Before concluding this section, one further complication bears mentioning. If the *BANK* variable is modified by another process while p's sequential operation is being executed, then it is possible for p to read inconsistent values from the *MEM* array. Observe that this does not result in p installing a corrupt version of the object, because p's subsequent SC fails. However, there is a risk that p's sequential operation might cause an error, such as a division by zero or a range error, because it reads an inconsistent state of the object. This problem can be solved by ensuring that, if *BANK* is invalidated, control returns directly from the *Read* procedure to the *LF_Op* procedure, without returning to the sequential operation. The Unix `longjmp` command can be used for this purpose. (We model this behavior using a `goto` statement.) This eliminates the possible error conditions mentioned above, and also avoids unnecessary work, because the subsequent SC will fail, and the operation will have to retry anyway.

The space overhead[4] (in terms of words) of the shared variables in the algorithm in Figure 4 is $\Theta(B + N \cdot T \cdot S)$, and the space complexity of the private variables for each of N processes is $\Theta(B + T)$. Thus, the overall space complexity of the algorithm is $\Theta(N \cdot B + N \cdot T \cdot S)$. Because this algorithm is lock-free and not wait-free, the worst-case time for an operation to complete is unbounded. However, it is interesting to compare the contention-free time complexity of lock-free algorithms. The *contention-free time complexity* is the time taken to complete one operation if no other process is executing an operation. (The definition of lock-freedom requires termination in this case.) As shown in Section 3, the time complexity of executing one *Long_Weak_LL*, one *Long_VL*, and one *Long_SC* operation, is $\Theta(B)$. Because each operation is assumed to modify at most T blocks, statement 5 is executed at most T times during one operation, and the loop at statement 17 executes at most T iterations. Thus we have the following theorem.

Theorem 1: Suppose a sequential object *OBJ* can be implemented in an array of B S-word blocks such that any operation modifies at most T blocks and has worst-case time complexity C. Then, *OBJ* can be implemented in a lock-free manner with space overhead $\Theta(N \cdot B + N \cdot T \cdot S)$ and contention-free time complexity $\Theta(B + C + T \cdot S)$. $\square$

These complexity figures compare favorably with those of Herlihy's lock-free construction. Consider the implementation of a queue. Using the arrangement shown in the example of Figure 5, an enqueue or dequeue operation can be performed in our construction by copying only two blocks: the block containing the head or tail pointer to update, and the block containing the array slot pointed to by that pointer. Thus, for our construction, $T = 2$, and space overhead is $\Theta(N \cdot B + N \cdot S)$. Contention-free time complexity is $\Theta(B+C+S)$, which is only $\Theta(B+S)$ greater than the time for a sequential enqueue or dequeue. In contrast, as mentioned earlier, each process in Herlihy's construction must actually copy the entire queue, even when using his large-object techniques. Thus, space overhead is at least N times the worst-case queue length, i.e., $\Omega(N \cdot B \cdot S)$. Also, contention-free time complexity is $\Omega(B \cdot S + C)$, since $\Omega(B \cdot S)$ time is required to copy the entire queue in the worst case. It might seem possible that our construction would suffer similar disadvantages for different objects. In fact, this is not the case, because using our construction, an operation copies a part of the object only if the sequential version of the operation modifies that part of the object.

[4]By *space overhead*, we mean space complexity beyond that required for the sequential object.

5 Wait-Free Universal Construction for Large Objects

In this section, we present our wait-free construction for large objects, which is shown in Figures 6 and 7. The basic structure of this algorithm is similar to that of the lock-free construction presented in the previous section. In particular, this algorithm provides the illusion of a contiguous array by maintaining a bank of pointers to the blocks that make up that array, and by allowing processes to perform operations on logical views that share blocks with the current array. As before, operations performed using this construction use the *Read* and *Write* procedures to access these logical views. Also, the mechanisms for using copy blocks and reclaiming displaced blocks are exactly the same as in the lock-free construction. However, as explained below, each process has sufficient copy blocks in this algorithm to perform the operation of at least one other process together with its own. Therefore, each process has $M \geq 2T$ private copy blocks. (Recall that T is the maximum number of blocks modified by a single operation.)

The principal difference between our lock-free and wait-free constructions is that processes in the wait-free construction "help" each other in order to ensure that each operation by each process is eventually completed. The overall structure of our helping mechanism is similar to that of Herlihy's helping mechanism [8]. Specifically, helping is achieved by having each process p "announce" its operation in $ANC[p]$ before entering a loop that attempts to perform its own operation, possibly together with the operations of other processes. This loop is repeated until either p executes a successful SC or p detects that its operation has been helped by another process. Despite the similarities in structure between our helping mechanism and Herlihy's, the details are quite different. The reason is that Herlihy's helping mechanism requires that a process be able to help the operations of all other processes at once, which may require excessive space. Our helping mechanism overcomes this problem because it only requires each process to have sufficient copy space to apply *two* operations (its own and one other). In the paragraphs below, we explain our helping mechanism in detail, as well as the mechanisms used for detecting termination and communicating return values.

Our helping mechanism. To facilitate helping in our wait-free construction, two new fields — *help* and *ret* — are added to the *BANK* variable of the lock-free construction presented earlier. The *help* field records the next process to be helped, and the *ret* field points to a block that contains operation return values and information that allows processes to detect completion of their operations.

We first describe the use of the new *help* field of *BANK*. In Herlihy's construction, each time a process performs an operation, it also performs the pending operations of *all* other processes. However, in our construction, the restricted amount of private copy space might prevent a process from simultaneously performing the pending operations of all other processes. (Recall that our construction is designed to obviate the need for each process to have enough space to copy the entire object.) Therefore, each process helps only as many other processes as it can without violating its space constraints. The *help* counter, which is used to ensure that each process is eventually helped, indicates which process should be helped next. A process performs its own operation (statement 36 of Figure 7) and helps operations of other processes (statement 40) by calling the *Apply* procedure, passing as a parameter the process to be helped. The *Apply* procedure is described in detail later. Each time process p performs an operation, p helps as many processes as its space constraints permit, starting with the process stored in the *help* field. This is achieved by helping processes until too few private copy blocks remain to accommodate another operation (statements 39 to 42). (Observe that the *Write* procedure increments *dcnt* whenever a new block is modified.) Process p updates the *help* field so that the next process to successfully perform an SC starts helping where p stops (statements 44 and 45). Because we assume that each process has enough copy space to accommodate at least two operations, each successful SC operation advances the *help* field by at least one process. Thus, if some process repeatedly fails to perform its own operation, the *help* field eventually ensures that the operation of that process is performed.

```
constant
    $N$ = number of processes
    $B$ = number of blocks in shared object
    $S$ = block size in words
    $T$ = maximum number of blocks modified by an operation
    $M$ = number of copy blocks per process ($M \geq 2T$)

type
    $anctype$ = record $op$: $optype$; $pars$: $paramtype$; $seq$: 0..2 end;
    $retblktype$ = array[0..$N-1$] of record $val$: $objrettype$; $applied$, $copied$: 0..2 end;
    $blktype$ = array[0..$S-1$] of $wordtype$;
    $banktype$ = record $blks$: array[0..$B-1$] of 0..$B+N*M-1$; $help$: 0..$N-1$; $ret$: 0..$N$ end;
    $tupletype$: record $pid$: 0..$N-1$; $op$: $optype$; $pars$: $paramtype$; $val$: $objrettype$ end

shared variable
    $BLK$: array[0..$B+N*M-1$] of $blktype$;                 /* Array and copy blocks */
    $ANC$: array[0..$N-1$] of $anctype$;                      /* Announce array */
    $RET$: array[0..$N$] of $retblktype$;                     /* Blocks for operation return values */
    $LAST$: array[0..$N-1$] of 0..$N$;                        /* Last $RET$ block updated by each process */
    $BANK$: $banktype$                                        /* Bank of pointers plus help counter and return block indicator*/

initially
    $BANK.ret = N \ \wedge \ BANK.help = 0 \wedge \ (\forall p :: ANC[p].seq = 0 \ \wedge \ RET[N][p].applied = 0 \ \wedge \ RET[N][p].copied = 0 \ \wedge$
    $(\forall n : 0 \leq n < B :: BANK.blks[n] = N*M+n \ \wedge \ BLK[N*M+n] =$ ($n$th block of initial object value))

private variable
    $copy$, $oldlst$: array[0..$M-1$] of 0..$B+N*M-1$; $ptrs$: $banktype$; $dirty$: array[0..$B-1$] of boolean; $dcnt$: 0..$M$; $rb$, $oldrb$:
    0..$N$; $match$, $a$, $seq$: 0..2; $applyop$: $optype$;   $applypars$: $paramtype$; $rv$: $objrettype$; $tmp$, $b$: 0..$N$; $done$, $loop$: boolean; $i$:
    0..$B-1$;  $side$: 0..1; $curr$: record $pid$: 0..$N-1$; $tag$: 0..1 end; $try$: 0..$N-1$; $m$: 0..$M-1$;  $j, h$: 0..$N-1$; $k$ : 0..$S-1$

initially $(\forall n : 0 \leq n < M :: copy[n] = pM+n) \ \wedge \ rb = p \ \wedge \ side = 0 \wedge \ ptrs.help = 0 \ \wedge \ try = 0 \ \wedge \ dcnt = 0 \ \wedge \ seq = 0$
```

Figure 6: Wait-free large object construction (continued in Figure 7).

Return blocks. To enable a process to detect that its operation has been applied, and to determine the return value of the operation, we use a set of "return" blocks. There are $N+1$ return blocks, $RET[0]$, ..., $RET[N]$; at any time, one of these blocks is "current" (and is indicated by the *ret* field in the *BANK* variable) and each process exclusively "owns" one of the other return blocks. The current return block contains, for each process p, the return value of p's most recent operation, along with two 3-valued control fields: *applied* and *copied*. Together with $ANC[p].seq$, these two fields determine the state of p's current operation (if any). When p is not performing an operation (i.e., p is between calls to *WF_Op*), the values of $ANC[p].seq$ and p's *applied* and *copied* fields in the current return block are all equal. When p announces a new operation (statement 22), it also increments (modulo 3) $ANC[p].seq$, thereby making it different from its *applied* and *copied* fields. As explained below, this indicates that p now has an outstanding operation. Each time a successful SC operation is performed (thereby applying one or more operations), a new return block is installed (that is, a new value is written to *BANK.ret*). The new return block is obtained by first making a copy of the previous one (see statement 31) and by then making the following changes: the *applied* field is copied to the *copied* field for each process (statements 33 through 35), and for each process p that has been helped, p's return value is recorded (statement 19) and $ANC[p].seq$ is copied to p's *applied* field (statement 20). The reasons for these modifications are explained below.

Applying operations. When a process q attempts to apply an operation of process p by calling $Apply(p)$, it first checks whether process p has an outstanding operation by comparing $ANC[p].seq$ to the current *applied* field for process p (statements 14 and 15). (More accurately, it compares $ANC[p].seq$ to p's *applied* field in a *copy* of the current return block. As mentioned above, this copy is made by process q in statement 31. In the correctness proof presented in [10], it is shown that, if p's *applied* field in this copy is different from that

```
procedure Read(addr: 0..$B*S-1$)
                                        returns wordtype
1:  v := BLK[ptrs.blks[addr div S]][addr mod S];
2:  if ¬VL(BANK) then goto 43 else return v fi


procedure Write(addr: 0..$B*S-1$; val: wordtype)
3:  blkidx := addr div S;
4:  if ¬dirty[blkidx] then
5:      memcpy(BLK[copy[dcnt]],
                BLK[ptrs.blks[blkidx]], sizeof(blktype));
6:      dirty[blkidx] := true;
7:      oldlst[dcnt] := ptrs.blks[blkidx];
8:      ptrs.blks[blkidx] := copy[dcnt];
9:      dcnt := dcnt + 1
    fi;
10: BLK[ptrs.blks[blkidx]][addr mod S] := val


procedure Return_Block() returns 0..N
11: tmp := Long_Weak_LL(BANK, ptrs);
12: if tmp ≠ N then return LAST[tmp]
13: else return ptrs.ret
    fi


procedure Apply(pr: 0..$N-1$)
14: match := ANC[pr].seq;
15: if RET[rb][pr].applied ≠ match then
16:     applyop := ANC[pr].op;
17:     applypars := ANC[pr].pars;
18:     rv := applyop(applypars);
19:     RET[rb][pr].val := rv;
20:     RET[rb][pr].applied := match
    fi
```

```
procedure WF_Op(op: optype; pars: paramtype)
21: seq := (seq + 1) mod 3;
22: ANC[p] := (op, pars, seq);
23: done := false;
24: b := Return_Block();
25: while ¬done ∧ RET[b][p].copied ≠ seq do
26:     if Long_Weak_LL(BANK, ptrs) = N then
27:         for i := 0 to $B-1$ do dirty[i] := false od;
28:         dcnt := 0;
29:         oldrb := ptrs.ret;
30:         ptrs.ret := rb;
31:         memcpy(RET[rb], RET[oldrb], sizeof(retblktype));
32:         if VL(BANK) then
33:             for j := 0 to $N-1$ do
34:                 a := RET[rb][j].applied;
35:                 RET[rb][j].copied := a
                od;
36:             Apply(p);
37:             try := ptrs.help;
38:             loop := false;
39:             while $dcnt + T \leq M$ ∧ ¬loop do
40:                 if try ≠ p then Apply(try) fi;
41:                 try := (try + 1) mod N;
42:                 if try = ptrs.help then loop := true fi
                od;
43:             LAST[p] := rb;
44:             ptrs.help := try;
45:             if Long_SC(BANK, ptrs) then
46:                 for m := 0 to $dcnt-1$ do
47:                     copy[m] := oldlst[m]
                    od;
48:                 rb := oldrb;
49:                 done := true
                fi
            fi
        fi;
50:     b := Return_Block()
    od;
51: b := Return_Block();
52: RET[b][p].copied := seq;
53: return RET[b][p].val
```

Figure 7: Wait-free large object construction (continued from Figure 6).

in the current return block, then q's subsequent SC will fail, so q will not incorrectly apply an operation.) If these two fields are different, then q performs p's operation (statement 18), records the return value of the operation in its copy of the return block (statement 19), and copies the value read from $ANC[p].seq$ to p's *applied* field in q's copy of the return block (statement 20). Later, if q's SC is successful, then q's copy of the return block becomes part of the current object state. Thus, p's operation is not subsequently reapplied by another process, because that process finds that p's *applied* field in the current return block equals $ANC[p].seq$.

An example that demonstrates the use of the return blocks is shown in Figure 8. Figure 8(a) shows ANC and the RET blocks in the initial state. (In the example there are two processes, so there are three RET blocks.) In this state, the current return block is $RET[2]$, as indicated by $BANK.ret$. The *seq* field of ANC is 0 for both processes, and the *applied* and *copied* fields of the current return block are also 0. This indicates that neither process has an outstanding operation. Also, process 0's return block is $RET[0]$, and process 1's is $RET[1]$.

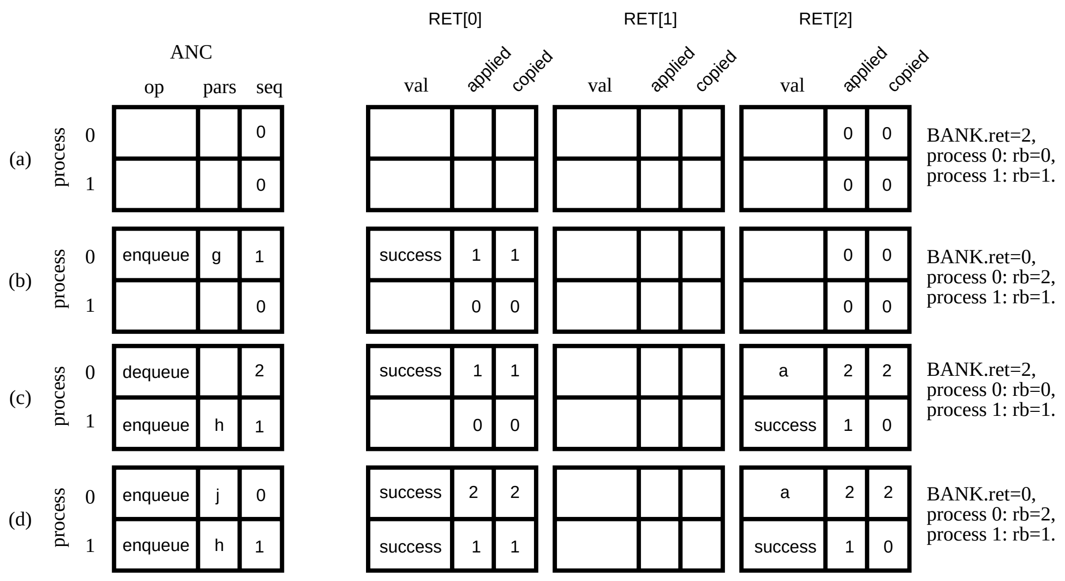

RET[0]
RET[1]
RET[2]
ANC
op
pars
seq
val
applied
copied
(a)
(b)
(c)
(d)
process
enqueue
dequeue
success
BANK.ret=2, process 0: rb=0, process 1: rb=1.
BANK.ret=0, process 0: rb=2, process 1: rb=1.
BANK.ret=2, process 0: rb=0, process 1: rb=1.
BANK.ret=0, process 0: rb=2, process 1: rb=1.

Figure 8: Example showing use of *applied* and *copied* fields.

Figure 8(b) shows the result of process 0 executing an *enqueue*(g) operation. Observe that process 0 has written its operation and parameters to $ANC[0]$ and has incremented $ANC[0].seq$ (statements 21 and 22). Process 0 has also copied the current return block into its own return block, performed its own operation, and made its own return block current. When it performed its own operation, process 0 copied $ANC[0].seq$ to $RET[0][p].applied$ (statement 20). Also, when a process successfully performs its own operation, it writes the value from the *seq* field of its *ANC* entry to its *copied* field in the current return block (statement 52). Process 0 has also reclaimed the previous return block ($RET[2]$) as its own return block for use in subsequent operations.

Figure 8(c) shows the result of process 0 executing a *dequeue* operation. Again, process 0 has announced and performed its own operation, and replaced the current return block with an appropriately modified copy. In this case, however, process 0 has also detected that process 1 has an outstanding *enqueue*(h) operation (by seeing that $ANC[1].seq \neq RET[0][1].applied$), and has performed this operation on behalf of process 1. As a result, process 1's entry in the new current return block ($RET[2]$) contains a return value for the enqueue operation, and also contains 1 in the *applied* field.

Finally, Figure 8(d) shows the result of process 0 executing another *enqueue* operation, this time with a parameter of j. This time, process 0 does not execute an operation for process 1 (because $ANC[1].seq = RET[2][1].applied$), but it does copy process 1's *applied* field of the return block to its *copied* field. As explained below, this allows process 1 to detect that its operation has been completed.

Detecting termination. The *copied* field for process q in the current return block is used by q to detect when its operation has been completed. In statements 33 to 35, the *applied* field is copied to the *copied* field for each process. Thus, the value in q's *copied* field of the current return block does not equal $ANC[q].seq$ until the successful SC *after* the one that applies q's operation (described above). To see why both *applied* and *copied* are needed to detect whether q's operation is complete, consider the two scenarios shown in Figure 9. In this figure, process 0 performs three operations. In the first operation, process 0's SC is successful, and process 0 replaces $RET[2]$ with $RET[0]$ as the current return block at statement 45. In Scenario 1, process 1 starts an operation during process 0's first operation. However, process 1 starts this operation too late to be helped by process 0. Before process 0's execution of statement 45, process 1 determines that $RET[2]$ is the current return block (statement 26). Now, process 0 starts a second operation. Because process 0

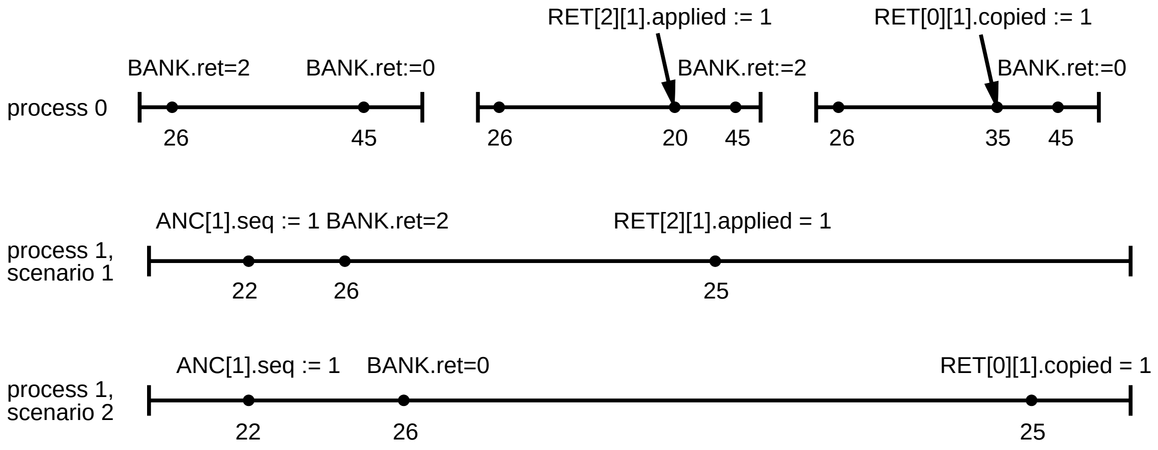


Figure 9: Scenario 1: process 1 prematurely detects that its *applied* field equals $ANC[1].seq$. Scenario 2: process 1 detects that its *copied* field equals $ANC[1].seq$. Points in time are labeled with statement numbers from Figure 7.

previously replaced $RET[2]$ as the current return block, $RET[2]$ is now process 0's private return block, so its second operation uses $RET[2]$ to record the operations it helps. Process 0 changes process 1's *applied* field to indicate that it has applied process 1's operation (statement 20). Note that, at this stage, process 1's operation has only been applied to process 0's private object copy, and process 0 has not yet performed its SC. However, if process 1 reads the *applied* field of $RET[2]$ at statement 25 instead of the *copied* field, then it incorrectly concludes that its operation has been applied to the object, and terminates prematurely. (Note that process 1 previously determined $RET[2]$ to be the current return block.) In the linearizability proof presented in [10], each operation is linearized to the successful SC that completes the operation. Therefore, the premature termination of process 1 would violate linearizability. As explained below, the use of the second field (*copied*) prevents this from happening.

As shown in Scenario 2 of Figure 9, it is similarly possible for process 1 to detect that its *copied* field in a return block equals $ANC[1].seq$ before the SC (if any) that makes that block current. However, because process 1's *copied* field is updated only *after* its *applied* field has been successfully installed as part of the current return block, it follows that some process must have previously applied process 1's operation. Thus, the use of the *copied* field ensures that process 1 terminates correctly.

Identifying the current return block. It remains to describe how process q determines which return block contains the current state of q's operation. It is not sufficient for q to perform a weak-LL on $BANK$ and read the *ret* field, because the weak-LL is not guaranteed to return a value of $BANK$ if a successful SC operation interferes. In this case, the weak-LL returns the identifier of a "witness" process that performs a successful SC on $BANK$ during the weak-LL operation. In preparation for this possibility, process p records the return block it is using in $LAST[p]$ (statement 43) before attempting to make that block current (statement 45). When q detects interference from a successful SC, q uses the $LAST$ entry of the witness process to determine which return block to read (statement 12). The $LAST$ entry contains the index of a return block that was current during q's weak-LL operation. If that block is subsequently written after being current, then it is a copy of a more recent return block, so its contents are still valid.

Suppose a sequential object OBJ whose return values are at most R words can be implemented in an array of B S-word blocks such that any operation modifies at most T blocks and has worst-case time complexity C. Then, the worst-case cost of one iteration of the loop at statements 25 to 50 is $\Theta(B+N(R+C)+M\cdot S)$. Also, because each sequential operation modifies at most T blocks, and because each process has M local copy blocks available, each successful operation is guaranteed to advance the help pointer by $\min(N, \lfloor M/T \rfloor)$.

Therefore, if process p's SC fails $\lceil N/\min(N, \lfloor M/T \rfloor) \rceil$ times, then p's operation is helped. Thus, we have the following theorem.

Theorem 2: Suppose a sequential object *OBJ* whose return values are at most R words can be implemented in an array of B S-word blocks such that any operation modifies at most T blocks and has worst-case time complexity C. Then, for any $M \geq 2T$, *OBJ* can be implemented in a wait-free manner with space overhead $\Theta(N \cdot (N \cdot R + M \cdot S + B))$ and worst-case time complexity $\Theta(\lceil N/\min(N, \lfloor M/T \rfloor) \rceil (B + N \cdot (R + C) + M \cdot S))$.[5] $\Box$

6 Performance Comparison

In this section, we describe the results of performance experiments that compare the performance of Herlihy's lock-free construction for large objects to our two constructions on a 32-processor KSR-1 multiprocessor.

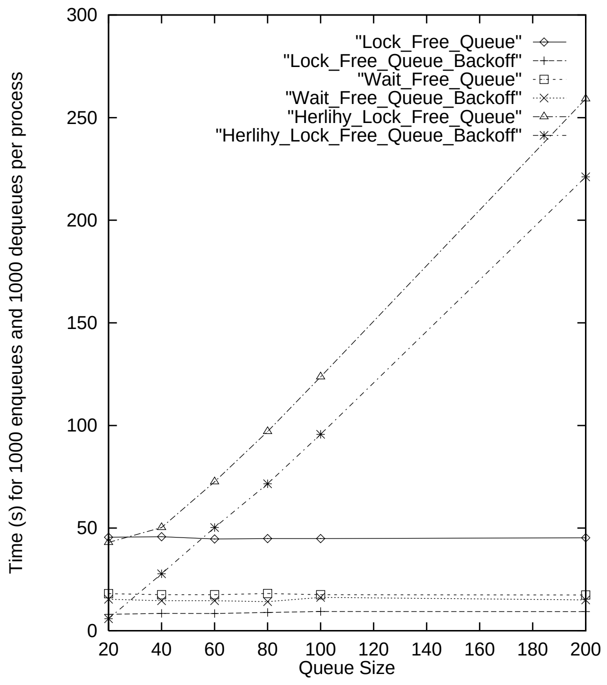
The results of one set of experiments are shown in Figure 10(a). In these experiments, LL and SC primitives were implemented using the standard spin-locking primitives provided by the KSR. Each of 16 processors performed 1000 enqueues and 1000 dequeues on a shared queue. Each point in the performance graphs presented in this section represents the average time taken to execute these operations over five runs. However, the variance in these times was small enough that taking these averages did not have a significant effect on the performance results presented.

Our large-object constructions give rise to a tradeoff between the block size S and the number of blocks B. Specifically, if S is large, then it is expensive to copy one block, but if S is small, then B must be large, which implies that the *Long_Weak_LL* and *Long_SC* procedures will be expensive. For testing our constructions, we chose B (the number of blocks) and S (the size of each block) to be approximately the square root of the total object size. This minimizes the sum of the block size and the number of blocks. (This is somewhat simplistic as we have not done extensive experiments to determine the relative costs of block copying and the *Long_Weak_LL* and *Long_SC* procedures. It is conceivable that further tuning of these parameters could result in better performance.) Also, we chose $T = 2$ because each queue operation accesses only two words. For the wait-free construction, we chose $M = 4$. This is sufficient to guarantee that each process can help at least one other operation. In fact, because two consecutive enqueue (or dequeue) operations usually access the same block, choosing $M = 4$ is sufficient to ensure that a process often helps all other processes each time it performs an operation. These choices for M and T result in very low space overhead compared to that required by Herlihy's construction.

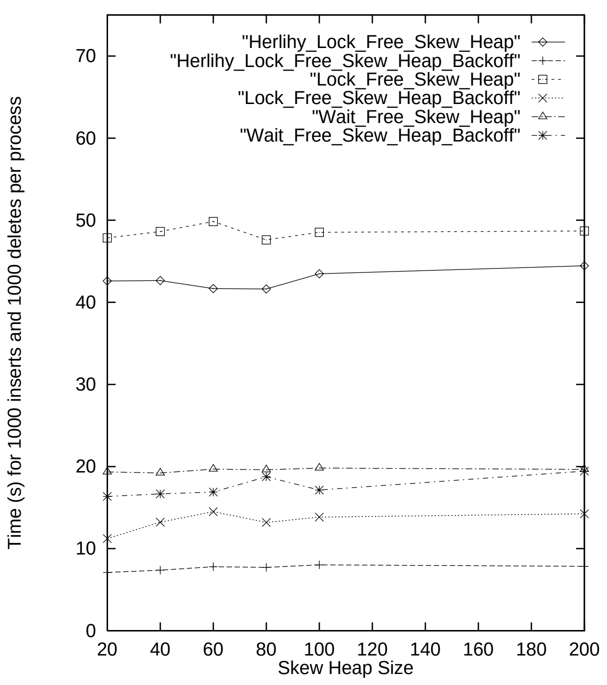
As expected, both our lock-free and wait-free constructions significantly outperform Herlihy's construction as the queue size grows. This is because an operation in Herlihy's construction copies the entire object, while ours copy only small parts of the object. It is interesting to note that our wait-free construction outperforms our lock-free one. We believe that this is because the cost of recopying blocks in the event that a SC fails dominates the cost of helping.

In Herlihy's performance experiments on small objects [8], exponential backoff played an important role in improving performance. Exponential backoff is implemented by introducing a random delay after each failed SC operation. The length of this delay is chosen from a uniform, random distribution between zero and a maximum delay. The duration of the maximum delay doubles (up to a set limit) with each successive failed SC, and is reset to a very small value at the beginning of each operation. The limit on the length of the maximum delay is an important parameter for achieving good performance: if it is set too low, then the benefits of backoff are not realized, and if it is set too high, processes can wait too long before retrying. The data shown for constructions with backoff represent the best performance we could achieve by tuning the backoff delay limit and repeating these experiments. This highlights the advantage of our

[5]When considering these bounds, note that for many objects, R is a small constant. Also, for many linear structures, including queues and stacks, C and T are constant, and for balanced trees, C and T are logarithmic in the size of the object. Finally, because $M \geq 2T$, $\lceil N/\min(N, \lfloor M/T \rfloor) \rceil$ is at most $N/2$.



(a)



(b)

Figure 10: Comparison of our implementation to Herlihy's on a KSR multiprocessor. (a) FIFO queue implementation. (b) Skew heap implementation.

wait-free construction, which, without resorting to using exponential backoff, outperforms the pure lock-free constructions, and performs comparably with the lock-free constructions with backoff.

We should point out that we deliberately chose the queue object to show the advantages of our constructions over Herlihy's. The queue represents an extreme case, for which Herlihy's construction necessarily copies the entire object, while ours do not. To consider an object more favorable to Herlihy's constructions, we also implemented a skew heap — the object considered by Herlihy in [8]. Skew heap operations usually modify only a few nodes near the root of a tree, and therefore avoid the long copying chains exhibited by Herlihy's construction when used to implement a queue.

As a first step towards implementing a skew heap, we implemented a dynamic memory allocation mechanism on top of our large object construction. This provides a more convenient interface for objects (including skew heaps) that are naturally represented as nodes that are dynamically allocated and released. There are well-known techniques for implementing dynamic memory management in an array. However, several issues arise from the design of dynamic memory management techniques in the context of our constructions. First, the dynamic memory allocation procedures must modify only a small number of array blocks, so that the advantages of our constructions can be preserved. Second, fragmentation complicates the implementation of *allocate* and *release* procedures. For example, after many allocate and release calls, the available free space can be distributed throughout memory, and it might be time-consuming, or even impossible, to find a contiguous block that is sufficiently large to satisfy a new allocate request. These complications can make the procedures quite inefficient, and can even cause the *allocate* procedure to incorrectly report that insufficient memory is available. Both of these problems are significantly reduced if the size of allocation requests is fixed in advance. For many objects, this restriction is of no consequence. This is true in the case of a skew heap, because all of the nodes in a skew heap are of the same size. We took advantage of this fact to simplify the design of our dynamic memory allocation library.

Having implemented dynamic memory allocation, we then implemented a large skew heap, and conducted performance experiments similar to those we conducted for the queue. The results of these experiments can be seen in Figure 10(b). As this figure shows, our constructions performed about the same as they did when used to implement a queue. (Note that the time scales on these two graphs are not the same.) However, Herlihy's construction performed much better than before, because unlike the queue operations, skew heap operations do not need to modify blocks that are at the end of long "chains" of blocks. In fact, Herlihy's lock-free construction slightly outperforms ours in this case. Recall that, in order to use Herlihy's construction, a programmer must determine, based on the semantics of the implemented object, which parts of the object must be copied by each operation. Thus, the implementation using Herlihy's construction is hand-crafted to perform exactly the right amount of copying; our construction does not rely on the programmer to provide this information. In other words, our construction sacrifices performance slightly in order to provide a transparent interface to the programmer.

7 Concluding Remarks

Our constructions improve the space and time efficiency of lock-free and wait-free implementations of large objects. Also, in contrast to similar previous constructions, ours do not require programmers to determine how an object should be fragmented, or how the object should be copied. However, they do require the programmer to use special *Read* and *Write* functions, instead of the variable references and assignment statements used in conventional programming. Nonetheless, as demonstrated by Figure 3, the resulting code is very close to that of an ordinary sequential implementation. Our construction could be made completely seamless by providing a compiler or preprocessor that automatically translates assignments to and from *MEM* into calls to the *Read* and *Write* functions.

The sequential code we used to implement operations for a skew heap object is based on dynamic allocation of heap nodes. To support this implementation, we also implemented a simple dynamic memory allocation mechanism. The applicability of our constructions can be further improved by providing the code for our dynamic memory allocation mechanism to programmers in a library. This would simplify the use of our constructions in implementing objects, such as trees, that are naturally implemented as blocks that are dynamically allocated and released.

One drawback of our constructions is that they are subject to a tradeoff between block size and the number of blocks. One of these must be at least the square root of the total object size. Furthermore, our constructions do not allow parallel execution of operations, even if the operations access disjoint sets of blocks. For example, in our shared queue implementations, an *enqueue* operation might unnecessarily interfere with a *dequeue* operation. In [2], we addressed similar concerns when implementing wait-free operations on multiple objects. More recently, Moir has developed new lock-free and wait-free constructions that support general transactions, are not subject to the tradeoff mentioned above, and allow parallel execution of transactions that do not conflict with each other [12]. While these constructions do in principle have some advantages over the ones presented here, they are somewhat more complicated. We therefore believe that our constructions will outperform the new ones in some applications, especially those in which objects do not admit much parallelism, or contention for objects is not sufficient to allow parallel execution of transactions to offer any advantage. Simulations carried out by Christopher Filachek [5] have confirmed that, at least for some objects, the constructions presented here outperform those presented in [12].

Acknowledgement: We would like to thank Lars Nyland for his help with the performance studies in Section 6.

References

[1] Y. Afek, D. Dauber, and D. Touitou, "Wait-free Made Fast (Extended Abstract)", *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995, pp. 538-547.

[2] J. Anderson and M. Moir, "Universal Constructions for Multi-Object Operations", *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 184-194.

[3] J. Anderson and M. Moir, "Universal Constructions for Large Objects", *Proceedings of the Ninth International Workshop on Distributed Algorithms*, 1995, pp. 168-182.

[4] G. Barnes, "A Method for Implementing Lock-Free Shared Data Structures", *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, pp. 261-270.

[5] C. Filachek, *Evaluation and Optimization of Lock-Free and Wait-Free Universal Constructions for Large Objects*, Master's thesis, Department of Computer Science, University of Pittsburgh, 1997.

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

[7] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp. 124-149.

[8] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 5, 1993, pp. 745-770.

[9] A. Israeli and L. Rappoport, "Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives", *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, August 1994, pp. 151-160.

[10] M. Moir, *Efficient Object Sharing in Shared-Memory Multiprocessors*, Ph.D. thesis, University of North Carolina at Chapel Hill, 1996. UMI Dissertation Services, 1996.

[11] M. Moir, "Practical Implementations of Synchronization Primitives", *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, 1997, pp. 219-228.

[12] M. Moir, "Transparent Support for Wait-Free Transactions", *Proceedings of the 11th Annual International Workshop on Distributed Algorithms*, 1997, pp. 305-319.

[13] N. Shavit and D. Touitou, "Software Transactional Memory", *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 204-213.

A Correctness of Wait-Free Algorithm

In this section, we give high-level, intuitive linearizability and wait-freedom proof sketches for our wait-free algorithm. For more rigorous and detailed proofs, the reader is referred to [10].

Linearizability [6] essentially requires that, in any execution, each operation "appears" to be applied to the shared object atomically at some point within its execution (i.e., after it is invoked, and before it returns). We prove that our algorithm has this property by defining a *linearization point* for invocations. When a process q performs a successful *SC* (within the *Long_SC* procedure), we linearize all the operations that q has applied since its previous call to *Long_Weak_LL*, in the order they were applied. (We say that *process q applies an invocation of process p* if it executes statements 16 through 20 with $pr = p$.) It is easy to see that this is the order in which operations are applied. However, it is not so easy to see that every operation is applied exactly once, or that it is applied within its execution. The following lemmas establish these properties. We prove these lemmas by inductively assuming that they have not been violated previously, and then showing that they hold for any given operation under this assumption. For convenience, we first introduce some notation.

$$\begin{aligned} ANC(p) &\equiv ANC[p].seq \\ AV(p) &\equiv RET[BANK.ret][p].applied \\ CV(p) &\equiv RET[BANK.ret][p].copied \end{aligned} \qquad \Box$$

Thus, $ANC(p)$ is p's "announced" sequence number, $AV(p)$ is p's *applied* field in the current return block, and $CV(p)$ is p's *copied* field in the current return block. We also use the notation $p.i$ to refer to statement i of process p. In the following lemmas, x is universally quantified over $\{0,1,2\}$.

Lemma 1: If p is between invocations, and $ANC(p) = x \ \wedge \ AV(p) = x \ \wedge \ CV(p) = x$, then this continues to hold until the execution of statement $p.22$, immediately after which $ANC(p) = (x+1) \bmod 3 \ \wedge \ AV(p) = x \ \wedge \ CV(p) = x$ holds.

Proof: It is easy to see that if statement $p.22$ is executed while $ANC(p) = x \ \wedge \ AV(p) = x \ \wedge \ CV(p) = x$ holds for some x, then $ANC(p) = (x+1) \bmod 3 \ \wedge \ AV(p) = x \ \wedge \ CV(p) = x$ holds afterwards. It remains to show that no other statement falsifies $ANC(p) = x \ \wedge \ AV(p) = x \ \wedge \ CV(p) = x$ while p is between invocations.

First, we observe that only statement $p.52$ can modify $RET[b][p]$ while $BANK.ret = b$. (All other modifications to *RET* blocks are made to blocks that are not current. Too see why, observe that when a process performs a successful *SC*, it "claims" the previously current return block for use as its next return block (see statements 29 and 48). Because this happens only upon a *successful SC*, each process always uses a distinct *RET* block in which to prepare values that will be current when and if this process executes a successful *SC*.) Thus, we need only consider the case in which a successful *SC* operation by some process q changes *BANK.ret*. In this case, because the *SC* succeeds, it follows that the current return block has not changed since q copied it (at statement 31). Therefore, q copied x to both the *applied* field and the *copied* field of $RET[rb][p]$, where rb is q's return block (which becomes current upon q's successful *SC*). Also, because these fields both have the same value, the *copied* field is not modified by statement $q.35$. Finally, by the inductive assumption that Lemmas 1 through 5 held for previous invocations, it follows that there was a successful *SC* during p's previous invocation (if there was such an invocation), which implies that q's *LL* must have occurred after p's previous invocation. This in turn implies that if q read $ANC[p].seq$ at statement $q.14$, then it read x, and therefore did not execute statements $q.16$ through $q.20$, and therefore did not modify $RET[rb][p]$ here either. Thus, when q's *SC* causes $RET[rb]$ to become the current block, the values of $AV(p)$ and $CV(p)$ do not change. $\Box$

Lemma 2: Process p cannot exit the loop at statements 25 through 50 while $ANC(p) = (x+1) \bmod 3 \ \wedge \ AV(p) = x \ \wedge \ CV(p) = x$ holds.

Proof: Process p can exit the loop at statements 25 through 50 either by executing a successful *SC* (see statement 49), or by detecting that its *copied* field in a *RET* block determined by *Return_Block* is equal to $ANC(p)$ (see statements 25 and 50).

In the first case, as in Lemma 1, p copies x to both p's *applied* and *copied* fields of its return block. Also, because p calls *Apply* with p as a parameter (statement 36), and because $ANC(p) = (x+1) \bmod 3$ and p's *applied* field is x, p executes statement $p.20$. Thus, when p's return block becomes current, $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = x \ \wedge\ CV(p) = x$ no longer holds, so the lemma holds in this case.

In the second case, p calls *Return_Block* to determine a return block b, which it then checks to see if $RET[b][p].copied = x$. *Return_Block* returns a block index that either is current or has been current "recently" (i.e., since p called *Return_Block*), or has been copied from a block that is current or has been current recently. To see this, note that if the *Long_Weak_LL* in *Return_Block* returns N, then the value returned by statement $p.13$ is the current block. In this case, because $CV(p) = x$ holds, p does not exit the loop. Otherwise, *Return_Block* returns the last *RET* block used by some process q that performed a successful *SC* while p was executing *Long_Weak_LL*. Whether this block was current, or was copied from a block that was current, p's *copied* field in this block contains x, so p does not exit the loop. (Note that it is possible that this block has been copied from the current block, but has not itself become current. This can happen if q has executed statement $q.43$, thereby making the block available as the "last" block used by q, but has either not reached the *SC*, or the *SC* has failed. Nonetheless, because this block has been copied from the current block, and because $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = x \ \wedge\ CV(p) = x$ holds, it is not possible that q has modified p's *copied* field in this block.) $\Box$

Lemma 3: If p is executing the loop at statements 25 through 50, and $ANC(p) = (x+1) \bmod 3 \wedge AV(p) = x \ \wedge\ CV(p) = x$ holds, then this continues to hold until some process that has applied p's operation performs a successful *SC*, after which $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = x$ holds.

Proof: As in Lemma 1, $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = x \ \wedge\ CV(p) = x$ can only be falsified by a successful *SC* making a new *RET* block current. Also, if a process q's *SC* is successful, then the values it copied to $RET[rb][p].applied$ and $RET[rb][p].copied$ at statement $q.31$ are the current values of $AV(p)$ and $CV(p)$, namely x. Thus, when q copied p's *applied* field to its *copied* field in statements $q.34$ and $q.35$, the *copied* field did not change. There are two cases. If q applied p's operation, then it also set $RET[rb][p].applied$ to $(x+1) \bmod 3$, in which case installing $RET[rb]$ as the current copy block establishes $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = x$, so the lemma holds. Otherwise, q did not apply p's operation, so installing $RET[rb]$ as the current copy block does not falsify $ANC = (x+1) \bmod 3 \ \wedge\ AV(p) = x \ \wedge\ CV(p) = x$. $\Box$

Lemma 4: If p is executing the loop at statements 25 through 50, and $ANC(p) = (x+1) \bmod 3 \wedge AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = x$ holds, then p does not return from *WF_Op* until $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = (x+1) \bmod 3$ holds.

Proof: First, observe that any successful *SC* executed by a process q while $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \wedge CV(p) = x$ holds establishes $ANC(p) = (x+1) \bmod 3 \wedge AV(p) = (x+1) \bmod 3 \wedge CV(p) = (x+1) \bmod 3$. This is because process q copies the current values of $AV(p)$ and $CV(p)$, namely $(x+1) \bmod 3$ and x, to p's *applied* and *copied* fields of q's new return block (statement $q.31$) and then copies the *applied* field to the *copied* field (statements $q.34$ and $q.35$). Furthermore, because $ANC(p) = AV(p)$, q does not apply p's operation, and therefore does not modify p's *applied* field at statement $q.20$. Thus, when q's *SC* succeeds, it establishes $ANC(p) = (x+1) \bmod 3 \wedge AV(p) = (x+1) \bmod 3 \wedge CV(p) = (x+1) \bmod 3$, as required.

Finally, observe that, before p returns from *WF_Op*, it first calls *Return_Block* in order to try to identify the current return block. Then, process p updates its *copied* field in this block to $(x+1) \bmod 3$. If this block is still the current block, then this establishes $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = (x+1) \bmod 3$, as required. Otherwise, there has been a successful *SC*, so, as argued above,

$ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = (x+1) \bmod 3$ has already been established. $\Box$

Lemma 5: If process q executes a successful *SC* having applied an operation by process p, then $ANC(p) \neq AV(p)$ holds immediately before it does so, and $ANC(p) = AV(p)$ holds immediately afterwards.

Proof: Process q applies process p's operation only if the value of $ANC(p)$ it reads at statement q.14 differs from the value of p's *applied* field that q copied from the current *RET* block (see statement q.15). Because process q's *SC* is successful, this value is the current value of $AV(p)$ throughout the interval between q's *LL* and q's *SC*. Thus, at the point when q executes statement q.14, $ANC(p) \neq AV(p)$. By Lemmas 2, 3, and 4, p does not return from *WF_Op* between this point and the point at which q executes its successful *SC*, and therefore $ANC(p)$ does not change in this interval. (Note that if some other process performed a successful *SC* in this interval, then q's *SC* would fail, a contradiction.) Thus, $ANC(p) \neq AV(p)$ holds immediately before q's *SC*. Also, because q copies the value read from $ANC(p)$ to p's *applied* field in its return block (statement q.20), $ANC(p) = AV(p)$ holds after the *SC*. $\Box$

Lemma 6: (Linearizability) Each invocation by each process is applied at most once; an invocation is applied only after it is invoked and before it returns; and if the invocation returns, then it is applied at least once.

Proof: Lemmas 1 through 4 show that each process cycles through three "phases". In the first phase, $ANC(p) = x \ \wedge\ AV(p) = x \ \wedge\ CV(p) = x$ for some $x \in \{0, 1, 2\}$ (note that this holds initially). Lemma 1 shows that this phase is ended only by process p invoking an operation, and that this begins the second phase, during which $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = x \ \wedge\ CV(p) = x$ holds. Lemma 2 shows that p cannot return from its invocation during the second phase. Lemma 3 shows that the second phase is ended only by some process successfully applying p's operation, and that this begins the third phase, during which $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = x$ holds. It follows that p's operation is linearized at least once before p returns from its invocation. Lemma 5 shows that an operation of process p can only be linearized at the end of the second phase, which implies that each operation is linearized at most once. Finally, Lemma 4 shows that process p's third phase ends and that the variables are initialized to begin the first phase for the next invocation (thus the above argument can be used inductively to show that all invocations are linearized exactly once). This completes our linearizability proof sketch. $\Box$

Lemma 7: (Wait-freedom) Every operation invoked by process p returns after a finite number of p's steps.

Proof: It is easy to see that if p performs a successful *SC*, then p's operation returns (see statement p.49). Therefore, let us assume that p's *SC* fails repeatedly. This implies that other processes repeatedly perform successful *SC* operations. Because of our assumption that each process has sufficient space to perform any two operations, each process that performs a successful *SC* executes the loop at statements 39 through 42 at least once. Thus, each successful *SC* increases (modulo N) the value of *BANK.help* by at least one (see statements 37, 41, and 44). Therefore, p can perform failing *SC* operations only a finite number of times before some process successfully applies p's operation. By Lemma 3, $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = x$ holds immediately after this occurs. As argued in the proof of Lemma 4, any successful *SC* executed by a process q while $ANC(p) = (x+1) \bmod 3 \wedge AV(p) = (x+1) \bmod 3 \wedge CV(p) = x$ holds establishes $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = (x+1) \bmod 3$. Thus, after p's next *SC* (whether successful or not), $ANC(p) = (x+1) \bmod 3 \ \wedge\ AV(p) = (x+1) \bmod 3 \ \wedge\ CV(p) = (x+1) \bmod 3$ holds. By Lemma 1, this continues to hold until p's next invocation. Therefore, after this point, whether *Return_Block* returns the current *RET* block, or a recently-current block, this block will have $(x+1) \bmod 3$ in p's *copied* field, so p will detect that its operation has been applied and return. This completes our wait-freedom proof sketch. $\Box$